Implementación dun Ambiente Aberto e Distribuído para o Desenvolvemento de Aplicacións de Control Industrial

Xoán Carlos Pardo Martínez

Departamento de Electrónica y Sistemas
Universidade da Coruña

UNIVERSIDADE DA CORUÑA

DEPARTAMENTO DE ELECTRÓNICA E SISTEMAS

TESE DE DOUTORAMENTO

IMPLEMENTACIÓN DUN AMBIENTE ABERTO E DISTRIBUÍDO PARA O DESENVOLVEMENTO DE APLICACIÓNS DE CONTROL INDUSTRIAL

AUTOR: D. XOÁN CARLOS PARDO MARTÍNEZ
DIRECTOR: D. RAMÓN FERREIRO GARCÍA

A CORUÑA, 2004

AGRADECEMENTOS

Aos meus pais e á miña irmá, a enerxía da miña vida.

Á miña aboa Ofelia, o seu xeito de ollarme faime invencíbel.

Á miña familia, cos que compartín máis ausencias que presencias, compensareivos.

A Marta, que gusta do Outono. Eu amo o Outono.

Aos meus amig@s, a cada un por unha razón e a tod@s pola mesma, síntome moi afortunado.

Ao meu director de tese, polo seu estímulo intelectual e a súa grande humanidade, algún día outro mundo será posíbel.

Aos meus compañeiros de departamento, que sempre tiveron unha cara amábel diante dos meus problemas, non sabedes canto agradezo a vosa paciencia.

A Woody Allen, el xa sabe por qué.

A !NUNCA MÁIS!

Índice

Lista de figuras

Lista de táboas

Glosario de abreviaturas

AFCET	Association Française des Sciences et Technologies de l'Information et des Systèmes
ANDECS	Analysis and Design of Controlled Systems
ANSI	American National Standards Institute
ANTLR	Another Tool for Language Recognition
ARS	Avec Recherche de la Stabilité
AST	Abstract Sintax Tree
CACE	Computer Aided Control Engineering
CACSD	Computer Aided Control System Design
CASE	Computer Aided Software Engineering
CIM	Computer Integrated Manufacturing
CIMOSA	CIM Open System Architecture
CLADP	Cambridge Linear Analysis and Design Programs
COCA	Component Oriented Control Architecture
CORBA	Common Object Request Broker Architecture
CSMP	Continuous System Modelling Program
CSSL	Continuous System Simulation Language
DAIS	Design Advisor for Implementing Systems
DCOM	Distributed Component Object Model
DDC	Direct Digital Control
DDE	Dynamic Data Exchange
DEDS	Discrete Event Dynamic Systems
DLL	Dynamic Link Library
DOS	Disk Operating System
E/S	Entrada/Saida
FBD	Function Block Diagram
GEMMA	Guide d'Étude des Modes de Marches et d'Arrêts
GNU	GNU's Not Unix!
GRAFCET	GRAphe Fonctionnel de Commande Etapes/Transitions
GRAI	Graphes de Résultats et Activités Interreliés
GIM	GRAI Integrated Methodology
HDCCS	Hierarchical Distributed Computer Control Systems
HMI	Human Machine Interface
ICAM	Integrated Computer Aided Manufacturing
IDEF	ICAM Definition Method
IDN	Integrated Design Notation
IEC	International Electrotechnical Commission
IL	Instruction List
ISO	International Organization for Standardization

JIT	Just In Time
LD	Ladder Diagram
MFC	Microsoft Foundation Classes
MIS	Management Information System
MMS	Manufacturing Message Specification
MOF	Meta Object Facility
OCL	Object Constraint Language
OLE	Object Linking and Embedding
OMT	Object Modelling Technique
OPC	OLE for Process Control
PC	Personal Computer
PCCTS	Purdue Compiler Construction Tool Set
PERA	Purdue Enterprise Reference Architecture
PiCSI	Process Control Systems Integration
PID	Regulador con acción Proporcional, Integral e Derivativa
PLC	Programmable Logic Controller
POU	Program Organization Unit
RdP	Rede de Petri
RdPI	Rede de Petri Interpretada
RNA	Rede de Neuronas Artificiais
RTC	Run To Completion
RTOS	Real Time Operating System
RTTL	Real Time Temporal Logic
SA/RT	Structured Analysis / Real Time
SADT	Structured Analysis and Design Technique
SCADA	Supervision, Control And Data Acquisition
SFC	Sequential Function Chart
SLICE	Subroutine Library for Control Engineering
SLICOT	Subroutine Library in Systems and Control Theory
SRS	Sans Recherche de la Stabilité
SSEE	Sistema Experto
ST	Structured Text
STL	Standard Template Library
TCCS	Timed Calculus for Communicating Systems
TCP/IP	Transmission Control Protocol/Internet Protocol
TIC	Target Independent Code
TML	Timed Modal Logic
TTM	Timed Transition Model
UML	Unified Modelling Language
VEE	Virtual Engineering Environment
VI	Virtual Instrument
VPU	Visual Pascal Unit
XML	Extensible Markup Language

Capítulo 1. Introducción

Este capítulo presenta o traballo realizado nesta tese de doutoramento, describindo brevemente a súa motivación, obxectivos e os métodos utilizados na súa realización. A estructura do capítulo é a seguinte: en (§1.1) indícase brevemente os motivos que deron lugar ao traballo realizado, así como os antecedentes históricos e as principais liñas de investigación relacionadas; en (§1.2) descríbense as características e obxectivos da solución aportada; en (§1.3) indícanse os métodos e ferramentas utilizados; e, finalmente, en (§1.4) resúmense os contidos dos demais capítulos.

1.1. Motivación

Na actualidade o desenvolvemento de "software" para a implementación de sistemas de control por computador é unha tarefa realmente complexa para a que os enxeñeiros de control requiren da asistencia de ambientes "software" especializados. Os factores que afectan á complexidade dos sistemas de control son múltiples, entre outros:

1. As tecnoloxías aplicadas nos procesos industriais son cada vez mais complexas.
2. As actividades de control automatizadas son cada vez mais numerosas e ocupan un nivel cada vez mais alto na xerarquía de control.
3. O número de posibilidades tecnolóxicas das que se dispón para a realización do sistema de control é cada vez maior.
4. A cantidade de aspectos tecnolóxicos alleos á enxeñería de control que é preciso considerar, principalmente os relacionados coas tecnoloxías da información e as telecomunicacións, increméntase a un ritmo que fai moi difícil 'estar ao día'.
5. Os sistemas incorporan un número cada vez maior de requisitos adicionais alén dos relacionados directamente co control, como a integración cos sistemas de información da empresa, interfaces gráficas avanzadas, xestión de modelos, simuladores, xestión de múltiples configuracións, portabilidade, etc.

Os ambientes "software" que proporcionan asistencia ao desenvolvemento de "software" para sistemas de control reciben o nome de ambientes CACE[1], e son definidos como [26]: '*a aplicación do modelado e simulación asistidos por computador para o deseño e implementación de sistemas de control [realimentados]*'. Estes ambientes caracterízanse por

[1] O termo CACE é utilizado na actualidade para indicar a evolución das ferramentas CACSD a ambientes que proporcionan asistencia integrada por computador en todas as fases do ciclo de vida dun sistema de control, de xeito semellante ás ferramentas CASE na enxeñeria do "software".

combinar métodos, técnicas e ferramentas de diferentes dominios co obxectivo de proporcionar asistencia durante todas as fases do ciclo de vida dun sistema de control, dende a súa identificación e modelado ata a súa implementación e operación. Idealmente deberían facilitar a utilización de formalismos familiares aos enxeñeiros de control durante a especificación, deseño, modelado e simulación do sistema, e automatizar o máximo posíbel os detalles relacionados coa obtención do código para a súa implementación en arquitecturas de control distribuídas e heteroxéneas. O papel destas ferramentas é semellante ao das ferramentas CASE dentro do campo da Enxeñería do Software, coa peculiaridade de que integran formalismos e técnicas propios da enxeñería de control, e van ser manexadas por usuarios que non son expertos en desenvolvemento de "software".

O desenvolvemento de ambientes CACE ten moitos aspectos en común co desenvolvemento de ambientes asistidos por computador noutras áreas da enxeñería, como poden ser o deseño de circuítos, a enxeñería de procesos de fabricación, a enxeñería do "software", etc. A base desta disciplina a forman os métodos de modelado e análise matemática da teoría de control clásica e, gradualmente, fóronse incorporando novos métodos e técnicas provenientes de diferentes campos, como por exemplo: métodos numéricos de optimización; cálculo simbólico; linguaxes de simulación; formalismos de especificación de DEDS, técnicas de "soft-computing" (redes de neuronas, lóxica difusa, algoritmos xenéticos); paradigmas e linguaxes de programación (orientación a obxectos, programación lóxica, programación visual); arquitecturas "software" distribuídas (CORBA, DCOM, sistemas multiaxente); etc. O desenvolvemento de sistemas de control por computador modernos convertiuse nun área multidisciplinar que require da utilización conxunta e coordinada de diferentes métodos, técnicas e ferramentas que permitan integrar diferentes puntos de vista dun sistema, como o da enxeñería de control, o da enxeñería do "software" ou o da enxeñería das telecomunicacións.

No resto deste apartado ofrécese unha breve introdución histórica a algúns dos avances producidos en diferentes areas que influiron a evolución dos sistemas de control por computador e os ambientes CACE, e descríbense algunhas das liñas que na actualidade dirixen a investigación neste campo.

1.1.1. Antecedentes históricos

A evolución das ferramentas CACE [92][93] está intimamente ligada á evolución parella da complexidade nos sistemas de control [141]. As primeiras aplicacións dos computadores en actividades de control datan de finais dos anos 50, nas que era utilizado para calcular os valores de referencia dos lazos de regulación. A funcionalidade do computador no sistema era reducida e estaba limitada á supervisión e, nalgúns casos, ao control deses valores de referencia. Nos comezos dos anos 60 conectouse o computador directamente ao proceso mediante os transductores axeitados e os lazos de control implementáronse directamente no computador, desprazando este progresivamente aos reguladores analóxicos. Esta configuración recibiu o nome de control dixital directo (DDC) e tiña a avantaxe de facilitar a implementación de esquemas de control nos que era precisa a interacción entre varios lazos. Nesta altura aínda non se dispoñía de ferramentas CACE, mais era posíbel que enxeñeiros de control con nocións de programación en ensamblador ou Fortran abordaran por si mesmos todas as fases de desenvolvemento do sistema, dende a súa concepción inicial ata a fase de operación e mantemento.

Nos finais dos 60, a aparición dos microprocesadores permitiu a aplicación masiva dos sistemas de control por computador e a comezos dos 70, a aparición das primeiras redes de comunicación entre computadores, propiciou a instalación dos primeiros sistemas xerárquicos

distribuídos de control por computador (HDCCS). Estes sistemas caracterízanse pola utilización de varios computadores interconectados entre si e organizados dende un punto de vista lóxico en niveis, formando unha xerarquía de control. Os computadores con maior capacidade de proceso ocupan os niveis mais altos da xerarquía, nos que se realizan funcións de supervisión e coordinación dos niveis inferiores; e de análise e asistencia á toma de decisións sobre o proceso. Nos niveis inferiores están os equipos conectados directamente ao proceso como minicomputadores, PLCs ou microcontroladores encargados do control local dacordo ás estratexias indicadas polos niveis superiores. Entre ambos niveis, e dependendo do sistema concreto, pode haber diferentes niveis intermedios que proporcionan interfaces de operador e diferentes graos de control, coordinación e supervisión en función das necesidades. No referente ás ferramentas, é na década dos 60 e primeira metade da dos 70 cando aparecen as primeiras librarías para álxebra lineal (EISPACK [69], LINPACK [50]), as primeiras linguaxes e ferramentas de simulación (Simula [42], CSMP [159], CSSL [160]), as primeiras linguaxes de programación estructurada (Algol [125], Pascal [175]), as redes de Petri (RdP) [137], e o que podería ser considerado como primeira xeración de ferramentas CACSD (UMIST [142], CLADP [110]), programas que proporcionaban asistencia para a realización de actividades específicas dentro do ciclo de desenvolvemento do sistema de control.

Os finais dos 70 e a década dos 80 estivo caracterizada pola aparición do computador persoal que, debido ao abaratamento de custes que supuña, foi aplicado en actividades de control, habitualmente non nas relacionadas co control directo pois non se dispuña das interfaces co proceso axeitadas, senón nas relacionadas coa supervisión, análise e interface gráfica. Popularizáronse os denominados sistemas SCADA que utilizaban computadores persoais, nos que se implementaba a interface de operador e as funcións de análise e supervisión, conectados a PLCs e microcontroladores encargados de realizar o control directo. Comezan tamén os primeiros esforzos por interconectar o sistema de control co sistema de información de xestión (MIS) da empresa. É nesta década na que aparecen as primeiras ferramentas CACSD implementadas sobre librarías de cálculo numérico (Ctrl-C [107], MatrixX[2] [171], Impact [140]) e os que poden considerarse como primeiros ambientes CACE (Federated CACSD [158], ECSTASY [122], GE-MEAD [162], CACE-III [90]). Tamén aparecen neste periodo ferramentas para o cálculo numérico, simbólico e a simulación (MATLAB[3] [119], Maple[4] [39], SLICE [156], CSIM[5] [152]), formalismos para a descrición formal de DEDS (StateCharts [78], Grafcet [1]); aproximacións síncronas ao deseño de sistemas reactivos (Esterel [24]); popularízanse os ambientes gráficos (MacOS[6], Windows[7]); e aparecen linguaxes que inclúen conceptos que facilitan a estructuración do "software" para manexar a súa complexidade (ADA [166], Smalltalk [82]).

A aparición dos buses de campo, a proliferación de redes de área local, o espectacular incremento nas capacidades de proceso dos computadores persoais, os esforzos de estandarización cara a facilitar a aparición de sistemas 'abertos', os avances tecnolóxicos en campos como as bases de datos, os sistemas operativos en tempo real, as arquitecturas "software" distribuídas, os dispositivos multimedia, os robots ou a intelixencia artificial son só algúns dos factores que veñen influenciando o desenrolo dos sistemas de control dende o comezo da década dos 90 ata a actualidade. A tendencia actual é a de distribuír a 'intelixencia'

[2] Matrixx®, LabView® e LookOut® son productos de National Instruments. http://www.ni.com/
[3] MatLab® e Simulink® son productos de The MathWorks. http://www.mathworks.com/
[4] Maple® é un producto da compañía Waterloo Maple. http://www.maplesoft.com/
[5] CSIM® é un producto da compañía Mesquite Software. http://www.mesquite.com/
[6] Mac OS® é un poducto da compañía Apple Computer. http://www.apple.com/
[7] Windows® é un producto da compañía Microsoft. http://www.microsoft.com/

do sistema sobre unha xerarquía lóxica de computadores, equipos de control (computadores industriais, PLCs, microcontroladores, robots, etc.) e dispositivos de campo (sensores e actuadores) 'intelixentes' conectados fisicamente entre si mediante un número limitado de buses de comunicacións. Exemplos desta tendencia son os sistemas multiaxente ou os sistemas holónicos, ambos baséanse na definición de entidades autónomas (axentes e holóns, respectivamente) que deben cooperar entre si para a realización de actividades complexas. Estes sistemas reciben a denominación de sistemas heterárquicos debido a que neles existe tanto unha organización xerárquica, parella aos niveis de decisión da estructura organizativa da empresa, como unha rede de entes autónomos cooperantes distribuídos sobre a rede física do sistema de control.

No referente ás ferramentas, neste periodo aparecen ferramentas para o desenvolvemento gráfico de sistemas de adquisición de datos (LabView[2] [56], HP VEE[8]) e sistemas SCADA (LookOut[2], Intouch[9]). Defínese o estándar IEC 61131-3 [86] para programación de PLCs, o que da lugar a que comecen a aparecer os primeiros ambientes 'abertos' de programación de PLCs (IsaGraph[10], CodeSys[11]). Popularizase o uso de MATLAB[3] coa incorporación de diferentes "toolboxes" que permiten a súa aplicación en múltiples campos e a inclusión do Simulink's Real Time Workshop, que permite xerar código para sistemas de tempo real a partires dos modelos deseñados co Simulink[2]. Desenvolvense ferramentas avanzadas de cálculo numérico, simbólico e simulación (SLICOT [20], LAPACK [5], Mathematica[12] [176], Ptolemy [32]); linguaxes (C++ [161], Eiffel [118]), metodoloxías (Booch [28], UML [29]) e ferramentas CASE (Rational Rose[13], Real Time Studio[14]) orientadas a obxectos; ambientes para o desenvolvemento de sistemas en tempo real (Tornado[15], QNX[16]); estándares para sistemas distribuídos (CORBA [126], DCOM [154]) e numerosas tecnoloxías provenientes do campo da Intelixencia Artificial (SSEE, RNAs, lóxica difusa, algoritmos evolutivos) son aplicadas ao desenvolvemento de sistemas de control.

Estes avances, xunto coa integración cada vez maior entre o sistema de control e o MIS, permitiron a aparición de paradigmas para a xestión de procesos de fabricación, como o CIM [170] ou o JIT [44], que están baseados no suporte proporcionado polas tecnoloxías da información e as comunicacións. Na actualidade as tecnoloxías relacionadas coa Internet están a ter unha aceptación crecente na implementación destes paradigmas xa que proporcionan un conxunto de estándares para o intercambio e visualización de información que facilitan a integración dos diferentes subsistemas da empresa.

1.1.2. Liñas de investigación

Algunhas das liñas de investigación relacionadas coa infraestructura "software" que da soporte aos ambientes CACE que na actualidade están recibindo maior atención son as seguintes:

---

[8] HP VEE® é un producto da compañía Hewlett-Packard. http://www.hp.com/
[9] InTouch® HMI é un producto da compañía Wonnderware. http://www.wonderware.com/
[10] IsaGraph® é un producto da compañía Altersys. http://www.altersys.com/.
[11] CodeSys® é un producto da compañía Smart Software Solutions. http://www.3s-software.com/.
[12] Mathematica® é un producto da compañía Wolfram Research. http://www.wolfram.com/
[13] Rational Rose® é un producto da compañía IBM Rational Software. http://www.rational.com/
[14] Real Time Studio® é un producto da compañía Artisan Software Tools. http://www.artisansw.com/
[15] Tornado® é un producto da compañía WindRiver Systems. http://www.windriver.com/
[16] QNX® Momentics é un producto da compañía QNX Software Systems. http://www.qnx.com/

- *Evolución cara a arquitecturas independentes das ferramentas.* Os primeiros ambientes CACE tiñan unha arquitectura dependente das ferramentas que integraban (Figura 1.1). Un supervisor implementaba os servicios precisos para manexar as peculiaridades de cada ferramenta e proporcionaba unha linguaxe de comandos común a todas elas. Os ambientes máis actuais eliminan esta dependencia utilizando estándares de modelado e proporcionando unha interface de acceso a servicios básicos comúns a través dos que se integran as diferentes ferramentas. Un exemplo deste tipo de arquitectura é o amosado na Figura 1.2, denominado 'modelo da torradora' [17].

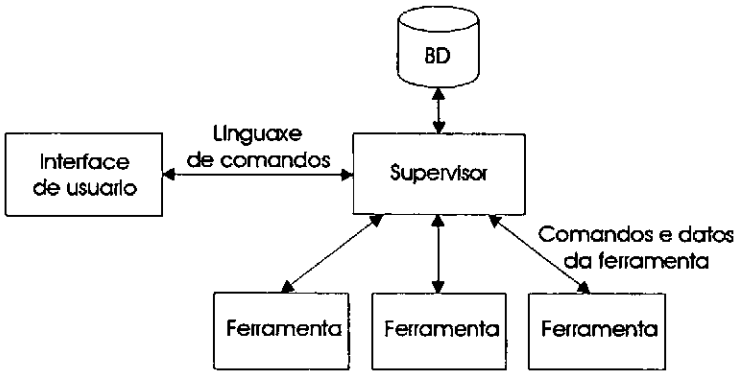


Figura 1.1: Arquitectura dunha ferramenta CACE dependente das ferramentas.

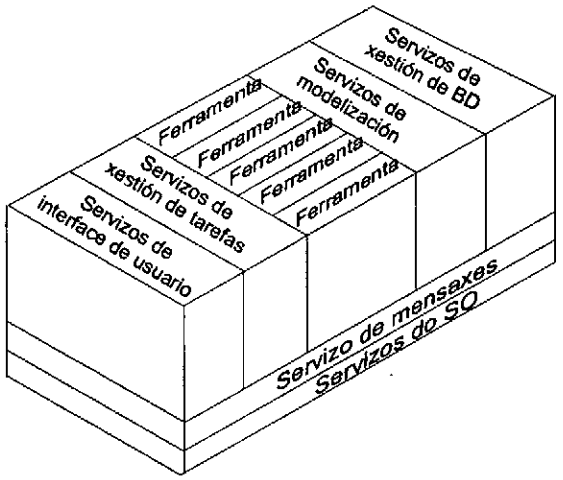


Figura 1.2: Arquitectura de referencia para ferramentas CACE independentes das ferramentas.

Algúns exemplos desta liña de investigación son:

1. O modelo unificado de información proposto en [168], que tenta dar unha definición neutral da información utilizada nun ambiente CACE.
2. A notación de deseño integrada (IDN) proposta en [17] para representar o modelo do sistema na ferramenta PiCSI. Esta notación consta de tres modelos relacionados: o de requirimentos, o arquitectónico e o modelo "software", que conxuntamente permiten modelar todos os aspectos implicados nas diferentes fases de desenvolvemento do sistema, dende a súa especificación ata a xeración de código. A IDN é unha notación de modelado orientada a obxectos que utiliza algúns dos diagramas propostos na UML.

3. Modelado mediante patróns de compoñentes ("template components"), como os utilizados na ferramenta de simulación Saber[17] ou os propostos en CIMOSA [169] para o modelado de empresas. Nesta aproximación os modelos particulares son creados a partires da instanciación de bloques xenéricos, o que aumenta a reusabilidade e aplicabilidade dos modelos.
4. A utilización do XML [151] para a descrición neutral da información e a linguaxe Java para a implementación dos servicios básicos comúns [17][93].

- *Desenvolvemento de sistemas de deseño virtual.* Estes sistemas manexan a información relacionada co proceso iterativo de deseño do sistema de control, asistindo na toma de decisións que permitan obter unha solución tendo en conta múltiples obxectivos e restriccións en conflicto. Algúns exemplos son:
  1. O traballo de [13] estudia a aplicación de patróns de deseño de "software" na implementación de sistemas "batch".
  2. A ferramenta ANDECS [71], unha aplicación de deseño multiobxectivo que asiste na síntese, simulación, análise e optimización do sistema de control.
  3. A ferramenta DAIS [163], un "front-end" intelixente que combina un ambiente CACE e unha ferramenta de desenvolvemento de sistemas expertos para implementar un sistema de asistencia ao deseño e implementación de sistemas de control.

- *Implementación de arquitecturas de integración* utilizando estándares para a computación distribuída. Algúns exemplos son:
  1. O uso de CORBA, máquinas virtuais para Java en tempo real e servidores Web para o intercambio da información do proceso no proxecto PiCSI [17].
  2. O uso de CORBA para implementar a especificación de paso de mensaxes ISO MMS [89] no proxecto COCA [70].

- *Xeración automática de código* a partires do modelo do sistema para a súa implementación en sistemas de tempo real. Esta é unha funcionalidade moi importante que permite illar ao enxeñeiro dos aspectos tecnolóxicos alleos á enxeñería de control implicados na implementación do sistema. É unha característica que incorporan un numero cada vez maior de ferramentas comerciais orientadas ao desenvolvemento de sistemas de control (Simulink Real-time Workshop, Matrixx/Autocode, LabView Application Builder) ou de "software" de propósito xeral (Rational Rose, Real Time Studio). As investigacións nesta liña non só procuran a xeración do código específico para cada arquitectura, senon tamén o desenvolvemento de técnicas para o mantemento da consistencia entre o modelo e o código xerado cando algún dos dous é modificado, a obtención do modelo a partires do código (enxeñería inversa), a verificación das condicións que garantan a seguridade no funcionamento do sistema ou o soporte á distribución das aplicacións.

- *A programación visual* permite construir e simular os modelos do sistema utilizando formalismos gráficos. A súa combinación con métodos formais de validación e con técnicas de xeración automática de código permite obter ferramentas de fácil manexo que reducen o esforzo de desenvolvimento dun sistema de control. Os seguintes son exemplos de ferramentas comerciais que incorporan esta característica:
  1. Aplicacións que utilizan UML como Rational Rose ou Real Time Studio.

[17] Saber® é un producto da compañía Synopsys. http://www.synopsys.com/.

2. Aplicacións que utilizan formalismos gráficos para a modelización de DEDS como o StateFlow en MATLAB, as RdP en Artifex[18] ou os StateCharts en Rational Rose.
3. Aplicacións que utilizan as linguaxes gráficas do estándar IEC 61131-3 como IsaGraph, PL7[19] ou CodeSys.
4. Aplicacións que utilizan linguaxes visuais propias como HP VEE ou LabView.

Ademais das liñas de investigación anteriores existen outras máis relacionadas cos métodos formais utilizados nos ambientes CACE, como por exemplo:

- A busca de novos métodos para a análise e síntese de sistemas de control mediante algoritmos numéricos, simbólicos ou "soft-computing".
- A proposta de novos métodos para a modelado e simulación de sistemas continuos, de eventos discretos ou híbridos.
- O estudio de novos métodos de optimización multiobxectivo para a análise e o deseño do sistema.
- O estudio de novas técnicas de supervisión e detección de fallas baseadas na utilización de modelos.
- A elaboración de estándares para o intercambio de información (principalmente os modelos do sistema e a información de deseño).

En resume, o desenvolvemento de ambientes CACE caracterízase por tratar con sistemas complexos que requiren de aproximacións multidisciplinares que integren métodos e técnicas procedentes de campos diferentes e nas que as tecnoloxías da informática e as telecomunicacións gañan cada vez maior importancia. A tendencia na actualidade é a de utilizar diferentes ferramentas especializadas que proporcionan soporte nas diferentes etapas do ciclo de vida do sistema de control, dende a súa concepción inicial ata a súa operación e mantemento. As ferramentas utilizadas comparten un modelo único do sistema representado mediante formalismos estándar e integranse a través dunha infraestructura "software" común.

1.2. Obxectivos

A finalidade do traballo realizado nesta tese de doutoramento é a de deseñar e implementar unha ferramenta que proporcione asistencia aos enxeñeiros de control no desenvolvemento de "software" para sistemas industriais e que poda ser integrada nun ambiente de desenvolvemento heteroxéneo, no que se combinen diferentes métodos e formalismos que proporcionen asistencia durante todo o ciclo de vida do sistema de control. En concreto a ferramenta proposta combina un formalismo gráfico para a especificación de controladores lóxicos secuenciais, o Grafcet, cunha aproximación orientada a obxecto. As funcionalidades que implementa inclúen a estructuración, modelado e simulación das secuencias lóxicas que describen o comportamento dos sistemas de control complexos, a integración de modelos desenvolvidos con outras ferramentas, e a xeración automática de código para a execución da aplicación en arquitecturas de control distribuídas e heteroxéneas. O desenvolvemento de "software" para a supervisión e control de DEDS e certos tipos de sistemas híbridos (p.e. sistemas "batch") son exemplos de posíbeis aplicacións da ferramenta.

[18] Artifex® é un producto da compañía Artis Software. http://www.artis.com/.
[19] PL7® é un producto da compañía Schneider Automation. http://www.modicon.com/.

A inclusión do Grafcet proporciona unha linguaxe gráfica normalizada, definida formalmente, fácil de aprender e utilizar, amplamente difundida e independente da tecnoloxía utilizada na súa implementación. As funcionalidades da ferramenta organízanse arredor da proposta e implementación dun metamodelo para este formalismo, e inclúen un editor gráfico e un compilador dos modelos representados utilizando o metamodelo proposto que permiten xerar automaticamente o código do sistema de control dende a representación gráfica da lóxica secuencial do seu funcionamento. A execución dos modelos compilados realízase nun intérprete Grafcet, no que pode escollerse a interpretación semántica desexada, e que forma parte dunha máquina virtual portábel na que pode configurarse dinamicamente a interacción do intérprete co proceso. O Grafcet é así utilizado de maneira uniforme durante a maior parte das fases do ciclo de vida do sistema de control, dende a especificación inicial ata a posta en funcionamento, operación e mantemento. Gracias ao soporte proporcionado pola ferramenta, o enxeñeiro utilízao para representar graficamente as secuencias lóxicas que modelan o comportamento do sistema, verifica a corrección sintáctica do modelo, valídao mediante simulación, xera automaticamente o seu código, execútao seguindo unhas regras semánticas ben definidas, depúrao e, finalmente, utilízao como interface de operador durante a operación do sistema.

O modelado, estructuración e reutilización das aplicacións implementadas coa ferramenta son reforzadas aproveitando as avantaxes da aproximación orientada a obxecto que proporcionan o metamodelo Grafcet proposto e a utilización da linguaxe C++ para a programación das accións de control e a representación da información utilizada nas aplicacións. O metamodelo Grafcet é definido como parte do metamodelo UML, de xeito semellante ao dos StateCharts, co que se facilita a integración da ferramenta con aplicacións CASE baseadas en UML, utilizando o Grafcet como alternativa para a especificación do comportamento dinámico dos elementos dos modelos. A linguaxe C++, ademais das propiedades de orientación a obxectos da propia linguaxe, permite a utilización nas aplicacións dun grande número de funcións incluídas en librarías externas, como por exemplo librarías matemáticas, de simulación, de "soft-computing" (redes de neuronas, lóxica difusa, algoritmos evolutivos), etc.

En conxunto a ferramenta proposta proporciona un ambiente de desenvolvemento "software" que utiliza un formalismo uniforme en diferentes fases do ciclo de vida dun sistema de control, dende o deseño inicial ate a operación e mantemento. As características do Grafcet combinan adecuadamente coas da orientación a obxectos na aplicación dun proceso de desenvolvemento iterativo, baseado na construcción de modelos gráficos executábeis que son validados e refinados progresivamente mediante simulación.

1.3. Métodos e ferramentas utilizados

No desenvolvemento da ferramenta proposta utilizouse unha aproximación orientada a obxectos [28] durante o deseño e a implementación. Aplicáronse diferentes patróns de deseño [67], considerando en especial algúns dos propostos para sistemas de tempo real [51][52] na implementación da máquina virtual. UML [29] foi a linguaxe de modelado utilizada e o C++ [161] a de programación. A aplicación de modelado UML utilizada foi o Rational Rose e o compilador o Visual C++, os dous na súa versión para o sistema operativo Windows. Para a programación gráfica do editor Grafcet utilizáronse as librarías MFC e Stingray Objective Studio. Esta é a única compoñente da ferramenta non portábel, debido a que se preferiu a utilización dunha libraría gráfica dependente do sistema operativo que ofrecera maiores posibilidades que as librarías gráficas portábeis. Ademais das ferramentas, librarías e métodos

indicados, que son ben coñecidos e non requiren maior explicación, utilizáronse outros que se describen a continuación.

1.3.1. SGI STL

Implementación libre da libraría STL [123][11] da empresa Silicon Graphics. A STL é unha libraría C++ xenérica que proporciona unha implementación baseada no uso de "templates" das estructuras de datos máis habituais (listas, colas, conxuntos, etc.) e os algoritmos que permiten manexalas.

1.3.2. Common C++

Libraría libre baixo licencia GNU que proporciona un conxunto de clases que facilitan a programación de aplicacións portábeis: "threads", "sockets", sincronización (semáforos, mutex), persistencia, carga dinámica de módulos compilados, etc. Estas clases abstraen os servizos básicos proporcionados polos sistemas operativos e eliminan parte dos problemas debidos ás diferencias entre eles. Na actualidade hai versións desta libraría para diferentes variantes de Unix e para Windows.

1.3.3. Sidoni

Sidoni [146] é unha aplicación para a simplificación de expresións booleanas extendidas, nas que ademais dos operadores do álxebra de Boole se utilicen os operadores de evento (↑ e ↓). Esta aplicación é utilizada durante a compilación dos modelos Grafcet para converter as expresións con eventos complexas, nas que os eventos afectan a expresións, en expresións con eventos simples, nas que os eventos afectan unicamente a variábeis.

1.3.4. PCCTS

PCCTS [135] é un conxunto de tres ferramentas para a xeración automática de analizadores sintácticos e traductores en C++: DLG, un xerador de analizadores léxicos; ANTLR, un xerador de analizadores sintácticos pred-LL(k), con k>1 (analizador descendente con predicados semánticos); e SORCERER, un traductor de árbores sintácticas abstractas (ASTs).

1.4. Sumario

Os contidos desta tese de doutoramento estructúranse da maneira seguinte:

- No Capítulo 2 preséntase as ideas principais nas que se basea o traballo realizado e descríbese a ferramenta proposta.
- No Capítulo 3 detállase a sintaxe e semántica do Grafcet, realízase unha comparación con outros formalismos, faise unha introducccción ao estándar IEC 61131-3 e ao SFC e mostranse algúns exemplos de modelado con Grafcet.
- No Capítulo 4 clasifícanse e analízanse diferentes tipos de ferramentas para o desenvolvemento de sistemas de control que utilicen algunha variante do Grafcet.
- No Capítulo 5 explícase en detalle o metamodelo proposto para o Grafcet, a libraría que o implementa e algúns exemplos da súa utilización.
- No Capítulo 6 descríbese a arquitectura e funcionamento do compilador que converte os modelos Grafcet ao formato utilizado para a súa carga dinámica e execución na máquina virtual.

- No Capítulo 7 detállase a arquitectura e implementación da máquina virtual que proporciona soporte á execución dos modelos Grafcet en ambientes distribuídos heteroxéneos.
- No Capítulo 8 detállase a implementación e funcionamento do intérprete de modelos Grafcet.
- No Capítulo 9 resúmense as conclusións e propóñense as futuras liñas de investigación.

Ademais tamén se inclúen algúns anexos con información de interese sobre diferentes aspectos relacionados co traballo realizado:

- No Anexo A inclúense as táboas do estándar IEC 61131-3 coas características definidas para o SFC.
- No Anexo B descríbense algunhas das funcionalidades básicas implementadas como parte da ferramenta desenvolvida nesta tese de doutoramento
- No Anexo C móstrase o arquivo *.mak* utilizado para compilar e enlazar co Visual C++ 6.0 as DLLs xeradas polo compilador Grafcet.
- No Anexo D móstrase a gramática ANTLR utilizada para xerar un analizador sintáctico de expresións C++ estendidas cos operadores de evento e temporización do Grafcet.
- No Anexo E móstranse as gramáticas Sorcerer utilizadas para a xeración automática de analizadores/traductores de expresións C++ estendidas cos operadores de evento e temporización do Grafcet.
- No Anexo F móstranse os patróns de comunicación e o intercambio de mensaxes para acceder aos servicios remotos da máquina virtual.

Debe facerse notar que, como se explica en (§2.3.4), unha das compoñentes da ferramenta proposta nesta tese de doutoramento é un editor gráfico a través do que se accede ás funcionalidades das outras compoñentes (compilador, máquina virtual, simulador). Para evitar que o tamaño desta documentación fose excesivo preferiuse non incluír nela un capítulo dedicado a describir o manexo deste editor, e decidiuse axuntalo como manual de usuario ao código e aos executábeis da ferramenta desenvolvida.

Capítulo 2. Fundamentos e descrición da ferramenta proposta

Neste capítulo preséntanse brevemente algúns conceptos básicos sobre os sistemas industriais e o proceso de desenvolvemento de "software" para o seu control por computador. Destácase o papel central dos modelos dentro deste proceso en aproximacións baseadas na utilización de compoñentes reutilizabeis, e sitúase a aplicación de metodoloxías de desenvolvemento "software" no contexto máis xenérico da enxeñería de sistemas industriais. Ademais preséntanse as funcionalidades básicas da ferramenta proposta nesta tese de doutoramento e compárase a aproximación adoptada con outras relacionadas.

O capítulo está organizado da forma seguinte: en (§2.1) dáse unha breve introducción aos sistemas industriais e ao seu control; en (§2.2) explícanse os aspectos relacionados co desenvolvemento de "software" para sistemas de control; as características, arquitectura e utilización da ferramenta proposta descríbense en (§2.3); e finalmente, o capítulo resúmese en (§2.4).

2.1. Os sistemas industriais e o seu control

Neste apartado descríbense as características que definen a un sistema industrial e introdúcense os conceptos básicos que se aplican no seu control. A descrición que se realiza e as definicións utilizadas están baseadas no estándar ISA S88.01 [87].

2.1.1. Descrición dun sistema industrial

Un sistema industrial pode definirse como [57]: '*o conxunto de equipamentos coa capacidade de realizar as operacións precisas para a obtención dun producto como parte da actividade de producción dunha empresa*'. Os sistemas industriais modernos están organizados como estructuras xerárquicas compostas por múltiples subsistemas que funcionan coordinadamente e se relacionan mediante diferentes fluxos de intercambio de materiais, enerxía e información. A actividade dun sistema industrial pode concibirse como un proceso de transformación de productos básicos en productos elaborados, no que se consume enerxía e xéranse residuos.

Os procesos realizados nun sistema industrial son secuencias de actividades físicas, químicas ou biolóxicas para a transformación, transporte ou almacenamento de materiais ou

enerxía [87]. Unha clasificación habitual dos procesos industriais divídeos en tres categorías dacordo á forma en que se obtén o producto:

1. *Procesos continuos*, nos que se obtén un fluxo continuo de producto.
2. *Procesos "batch"*, nos que o producto se fabrica por lotes ou coadas.
3. *Procesos de fabricación ou ensamblaxe de compoñentes*, nos que se obteñen cantidades discretas de producto.

Na práctica un sistema industrial está formado por unha combinación de procesos, e aínda que estea caracterizado principalmente pola presencia maioritaria dos pertencentes a unha das categorías anteriores, con frecuencia inclúen tamén procesos pertencentes ás outras.

Os equipamentos que forman parte dun sistema industrial poden clasificarse funcionalmente en dúas categorías: o equipamento de proceso e o equipamento de operación. O *equipamento de proceso* fórmano os dispositivos físicos utilizados para transportar, transformar e almacenar os productos e a enerxía; mentres que *o equipamento de operación* está formado polos dispositivos que asisten ao persoal no manexo do equipamento de proceso. Como parte do equipamento de operación poden considerarse tanto os sistemas de control como os de información de xestión da empresa.

Os equipamentos de proceso e control agrúpanse dende un punto de vista funcional para formar *unidades de proceso* [87]. Cada unidade pode realizar unha ou varias operacións e habitualmente estará composta por un elemento de proceso relevante (un decantador, un depósito, un reactor, etc.) e o equipamento auxiliar para o seu manexo (válvulas, motores, reguladores, etc.). A Figura 2.1 mostra a estructura xenérica dunha unidade de proceso na que se combinan equipamentos de proceso e control e se intercambian productos, enerxía e información con outras unidades. A estructura física dun sistema industrial poden representarse mediante a conexión de múltiples unidades de proceso como a mostrada. Ao conxunto de unidades que funcionan coordinadamente para a producción dun ou varios productos se lle denomina *cela de producción*.

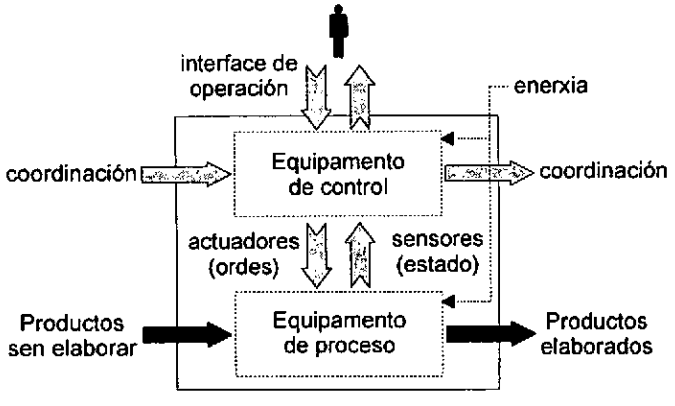


Figura 2.1. Estructura dunha unidade de proceso.

Dende o punto de vista operativo, o comportamento dos equipamentos de proceso e control é descrito mediante diferentes *estados de operación* (STARTING, RUNNING, PAUSED, etc.) representados utilizando algún tipo de diagrama de estados e transicións. O estado dun equipamento determina a forma en que este funciona e como reacciona aos comandos externos e condicións internas do proceso. Un exemplo de proposta para a formalización dos estados operativos dun equipamento industrial é o Gemma [121], que ten unha relación moi estreita co Grafcet [30].

2.1.2. Os sistemas de control industrial

Os sistemas de control caracterízanse por interactuar directamente co equipamento de proceso mediante o intercambio de ordes e informacións, sendo a combinación dambos a que proporciona a funcionalidade que permite obter un producto cumprindo coas especificacións de producción indicadas en cada momento. Un sistema de control moderno non se limita á regulación das magnitudes físicas do proceso, senón que coa aplicación crecente dos computadores e as tecnoloxías da comunicación, pasou a ser unha compoñente chave dos sistemas de información e xestión da producción da empresa. En [57] indícase que na actualidade o control de procesos consiste '*na integración funcional do control secuencial e de regulación clásicos coa xestión de información en tempo real*' e clasifica en tres categorías as funcións que realiza un sistema de control moderno:

1. A aplicación das estratexias de control da empresa.
2. O acceso á información sobre o proceso de producción para a asistencia na toma de decisións de xestión a medio e longo prazo.
3. O manexo do sistema que permita aos operadores a supervisión do sistema e a adopción de accións correctivas en caso de ser preciso.

Para dar soporte a estas funcións o sistema de control ademais de co proceso terá que interactuar cos operadores, co sistema de información da empresa e, en caso de ser un sistema distribuído, as diferentes partes que o forman tamén interactuarán entre si (Figura 2.1). A súa arquitectura lóxica adopta unha estructura xerárquica parella aos niveis de decisión (Figura 2.2) da empresa, na que os niveis máis baixos son os implicados máis directamente no control de procesos mentres que os mais altos son os que proporcionan asistencia na toma de decisións en actividades como a planificación da producción, o mantemento das instalacións, o control de calidade, a xestión de "stocks", etc.

Figura 2.2. Niveis lóxicos nun sistema de control industrial.

Ao nivel de proceso poden utilizarse diferentes tipos de control, cuxo funcionamento coordinado é o que proporciona as funcións que permiten aplicar as estratexias definidas pola empresa. Os *tipos de control* son independentes da tecnoloxía utilizada, por exemplo, un tipo de control básico é a regulación das magnitudes do proceso, que antes da aparición dos reguladores dixitais facíase utilizando a tecnoloxía electromecánica ou pneumática existente no momento. O avance da tecnoloxía de control permite mellorar as prestacións e a fiabilidade dos tipos de control, facilitar a súa instalación e o seu manexo, reducir custes, incrementar a cantidade e a complexidade das funcións dispoñíbeis, e aumentar o número de actividades automatizadas, principalmente nos niveis superiores da xerarquía de control. En [87] clasifícanse os tipos de control utilizados nun sistema industrial en tres categorías:

1. *O control básico*, que é o utilizado para activar e manter un estado determinado nun equipamento ou proceso. O control básico inclúe:
   a. *A regulación de magnitudes* do proceso arredor dun valor de consigna, realizada mediante os clásicos lazos de control xeralmente implementados mediante reguladores PID e nos que se utilizan diferentes configuracións ben coñecidas: regulación simple, en cascada, por adianto, de relación, selectivo ("override"), etc.
   b. *O control discreto* ("ON/OFF"), realizado mediante relés e utilizado para producir un cambio no estado do equipamento de proceso (acendido/apagado de motores, apertura/peche de válvulas, etc). O diagrama de contactos é a representación gráfica utilizada tradicionalmente para expresar a lóxica booleana do control discreto. Os PLCs foron inicialmente deseñados coa finalidade de executar a lóxica dos diagramas de contactos, substituíndo así ás implementacións cableadas ou pneumáticas anteriores.
   c. *O control secuencial*, baseado no control discreto é aplicado en procesos cuxa lóxica de funcionamento consiste na realización de operacións organizadas en secuencias ordenadas de pasos. Este é o tipo de control que habitualmente se utiliza en procesos de fabricación, ensamblaxe e almacenamento de pezas, no control da secuencia de fabricación en procesos "batch" e no control do arranque e a parada de procesos continuos.
   d. *A monitorización*, que consiste na obtención de información sobre o estado e evolución do proceso e os eventos que se detecten.
   e. *O manexo de excepcións*, que consiste na automatización das accións a realizar para recuperar un proceso cando se detecta un evento ou condición anormal.
2. *O control procedemental*, que describe as accións a realizar en cada unidade de proceso para a obtención dun producto mediante unha secuencia ordenada de operacións e fases. O estándar ISA S88.01 [87] define un diagrama que describe os estados (IDLE, STARTING, RUNNING, etc.) e as condicións de transición válidas (START, STOP, ABORT, etc.) para xestionar o funcionamento de procedementos, operacións e fases. A definición completa de cada procedemento, operación ou fase realízase especificando a lóxica secuencial das accións a aplicar en cada estado. Este tipo de control é o que habitualmente se utiliza en procesos "batch" e é fundamental nos sistemas de fabricación flexíbeis e nos esquemas de control baseados no uso de 'receitas'.
3. *O control de coordinación*, que é utilizado para controlar a utilización dos recursos compartidos por diferentes procesos ou 'lotes' de producción. Este tipo de control é utilizado habitualmente para coordinar a transferencia ou transporte de materiais entre diferentes unidades de producción en sistemas de fabricación flexíbeis.

2.1.3. Modelado de procesos industriais

No desenvolvemento de sistemas de control ten un papel moi importante o modelado do proceso mediante formalismos que permitan verificar as súas propiedades formais e estudiar o seu comportamento, xa sexa mediante métodos analíticos ou por simulación. Dende o punto de vista da súa dinámica, os modelos dos procesos industriais dividíronse tradicionalmente en dúas categorías:

1. *Os modelos continuos*, nos que o tempo é considerado como unha variábel continua. Os modelos matemáticos destes sistemas están baseados nos principios fundamentais de conservación da materia, o momento e a enerxía ou na observación empírica das relacións entre as magnitudes do sistema. Matematicamente son representados mediante sistemas de ecuacións alxebraicas ou diferenciais.

2. *Os modelos discretos*, nos que o tempo é considerado en instantes de tempo concretos. Matematicamente son representados mediante sistemas de ecuacións en diferencias.

Unha aproximación habitual no control de procesos é a de representar o sistema de control como un sistema dinámico de eventos discretos (DEDS), e utilizar algún dos numerosos formalismos existentes para o modelado, simulación e verificación de DEDS: os autómatas de estados finitos [104], os diagramas de estados ("StateCharts", [78]), as RdP [137][155], o Grafcet [46], etc. Os DEDS son representados mediante un conxunto discreto de estados a través dos que a situación do sistema evoluciona dirixida por eventos. Dependendo de se os eventos poden producirse en calquera momento ou unicamente en instantes concretos os DEDS clasifícanse en asíncronos e síncronos respectivamente.

As liñas de investigación máis recentes [8] céntranse nas *aproximacións híbridas*, que combinan os modelos continuos ou discretos do proceso coas aproximacións baseadas en modelos DEDS do sistema de control. Son numerosos os exemplos de fenómenos que son mellor representados mediante aproximacións híbridas: o arranque e parada de procesos continuos, os sistemas "batch" nos que se combinan procesos discretos e continuos, a utilización de reguladores dixitais en procesos continuos, etc. As principais motivacións para utilizar aproximacións híbridas no modelado de procesos industriais son:

1. A reducción da complexidade dos modelos con técnicas como por exemplo:
   a. A aproximación do modelo non lineal dun proceso continuo mediante varios modelos lineais entre os que se conmuta en función de determinadas condicións.
   b. A estructuración xerárquica do sistema de control utilizando modelos abstractos nos niveis superiores que son simplificacións discretas dos modelos continuos dos niveis inferiores.
2. A utilización de dispositivos dixitais para o control de procesos continuos é mellor modelada como un sistema híbrido, no que se utiliza un modelo continuo ou discreto para o proceso e un modelo DEDS para o sistema de control.

2.1.4. A arquitectura física do sistema de control

A Figura 2.3 mostra a arquitectura física utilizada como referencia neste traballo para proporcionar soporte á execución do "software" do sistema de control. Esta arquitectura está formada por computadores e PLCs nos que se dispón de sistemas operativos con capacidades de tempo real (RTOS) e que se interconectan mediante redes de comunicacións industriais. A interacción co proceso realízase mediante dispositivos de E/S interconectados utilizando buses de campo. En [35] analízase a idoneidade desta arquitectura para a implementación da estructura xerárquica dun sistema de control industrial como o da Figura 2.2.

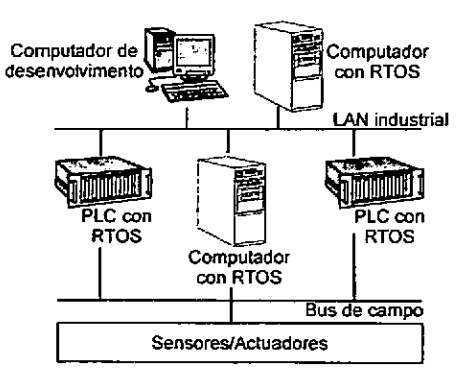


Figura 2.3. Arquitectura física dun sistema de control industrial.

2.2. O desenvolvemento de "software" para sistemas industriais

Neste apartado descríbense brevemente as características do proceso de desenvolvemento de "software" para sistemas de control industrial como parte da actividade desenvolvida no contexto máis xenérico da enxeñería de sistemas industriais. A descrición que se realiza e as definicións utilizadas están baseadas no estándar ENV 40 003 [34] e en traballos relacionados.

2.2.1. As arquitecturas de referencia na enxeñería de sistemas industriais

O desenvolvemento de "software" en sistemas industriais (e máis especificamente, o de "software" de control) debe entenderse como unha máis das actividades realizadas durante o ciclo de vida do sistema industrial, e as metodoloxías de desenvolvemento "software" como un método a ser integrado cos procedentes doutras disciplinas (como a enxeñería de procesos industriais ou a enxeñería de control, por exemplo) no contexto da aplicación das tecnoloxías da información e as comunicacións á enxeñería de sistemas industriais.

A enxeñería de sistemas industriais está sufrindo na actualidade un proceso semellante ao que no seu día se deu no campo da enxeñería do "software", coa busca de metodoloxías, métodos e ferramentas que permitan, mediante a utilización de ambientes asistidos por computador, a aplicación de aproximacións sistemáticas coas que podan manexarse as crecentes demandas dos novos paradigmas de fabricación (como o JIT [44] ou a enxeñería concorrente [136], por exemplo) derivados da necesidade de adaptarse a un mercado en continua e rápida evolución.

Debido á complexidade deste proceso nunha actividade que integra múltiples disciplinas consideradas dende diferentes puntos de vista (dende o técnico ate o económico ou xurídico) propuxéronse varias arquitecturas de referencia para o modelado de empresas[20] (GRAI/GIM [54], CIMOSA [4] ou PERA [174]). Estas arquitecturas proporcionan un marco de referencia no que situar os modelos, métodos e ferramentas utilizados ao longo do ciclo de vida da empresa, co obxectivo de obter un ambiente de modelado asistido por computador no que os modelos podan ser editados, executados e utilizados en último termo para a operación, monitorización e control das actividades da empresa. As arquitecturas propostas organízanse dacordo a tres dimensións [34]:

1. *A dimensión da xenericidade*, que é o nivel de abstracción correspondente aos elementos da arquitectura utilizados:
   a. *O nivel xenérico*, no que se definen os elementos de modelado básico.
   b. *O nivel parcial*, formado por modelos parciais que poden ser reutilizados e adaptados polo usuario para obter modelos particulares.
   c. *O nivel particular*, no que se describen os aspectos particulares da empresa utilizando os elementos do nivel xenérico e parcial.

   O proceso de particularización de modelos a partir dos elementos xenéricos e modelos parciais denomínase *refinamento por particularización* ou *instanciación*.

2. *A dimensión dos modelos*, que é o nivel de abstracción correspondente as fases do proceso de desenvolvemento do modelo da empresa:
   a. *Os modelos de requirimentos*, que definen as operacións a realizar pola empresa.

[20] O concepto de empresa é utilizado nestas arquitecturas nun sentido amplo, que inclue ao de sistema industrial definido anteriormente (§2.1.1).

b. *Os modelos de deseño*, que especifican como van a realizarse as operacións para cumprir cos requirimentos da empresa.
c. *Os modelos de implementación*, que describen os medios e regras a utilizar na execución das operacións da empresa.

O proceso de obtención dos modelos de implementación a partir dos de requirimentos e deseño denomínase *refinamento por derivación.*

3. *A dimensión das vistas*, que son descricións que se concentran en aspectos particulares da empresa co obxectivo de reducir a complexidade:
   a. *A visión funcional*, que proporciona unha descrición xerárquica das funcións, do comportamento e da estructura funcional da empresa.
   b. *A visión da información*, que proporciona unha descrición estructurada dos obxectos da empresa identificados nas outras vistas.
   c. *A visión dos recursos*, que proporciona unha descrición da organización dos recursos da empresa.
   d. *A visión organizativa*, que proporciona unha descrición da estructura organizativa da empresa.

   O número de vistas non é fixo e as vistas non son modelos en si mesmas senón diferentes puntos de vista do modelo da empresa. O proceso da súa obtención para cada nivel de modelado denomínase *refinamento por xeración.*

Unha das características definitorias destas arquitecturas é o papel central que adquire a actividade de modelado e o interese por conseguir a infraestructura que permita dispor de modelos parciais estandarizados que podan ser reutilizados adaptándoos aos aspectos particulares de cada sistema. Isto permitiría avanzar cara a unha situación equiparábel á que se da noutras áreas como a electrónica ou a enxeñería do "software" baseada en compoñentes: a existencia de ambientes abertos que posibiliten o desenvolvemento de arquitecturas consistentes, modulares e flexíbeis mediante a conexión de módulos estandarizados dispoñíbeis en forma de librarías ou módulos "software" precompilados (aproximación "plug-and-play" [139]).

2.2.2. O modelado na enxeñería de sistemas industriais

Como se indicou anteriormente o modelado é unha actividade central na enxeñería de sistemas industriais. Os modelos son utilizados para a análise, deseño, simulación e operación (control, coordinación e monitorización) dos sistemas, ademais de servir como medio de representación do coñecemento ("know-how"), como ferramenta de apoio á toma de decisións ou como elemento de integración de puntos de vista multidisciplinares, por exemplo. O modelado dun sistema industrial abrangue numerosos aspectos que deben ser considerados dende diferentes puntos de vista (funcional, información, recursos, organización, etc.) como, por exemplo: os procesos e a súa estructura funcional, os productos, os equipamentos de proceso e control, a estructura organizativa e de toma de decisións, etc.

En [169] descríbense os principios que se aplican no modelado de empresas (e sistemas industriais) para manexar a súa complexidade:

1. *Principio de separación temática*, que establece que a empresa non debe considerarse un todo, senón ser analizada por partes.

2. *Principio de descomposición funcional*, que establece que a empresas son sistemas dinámicos complexos definidos principalmente en base á súa funcionalidade que é modelada mediante unha xerarquía de subfuncións. A aproximación funcional de métodos como SADT [144] ou IDEF [81] é a utilizada tradicionalmente nos sistemas industriais. Recentemente propúxose en CIMOSA [4] a utilización dunha aproximación baseada en procesos.
3. *Principio de modularidade*, que establece que os modelos deben ser modulares, construídos mediante a ensamblaxe de bloques básicos predefinidos.
4. *Principio da xenericidade do modelo*, que establece a importancia de definir bloques xenéricos estandarizados que abstraian os aspectos comúns a diferentes dominios que podan ser adaptados para a súa utilización en aplicacións particulares.
5. *Principio de reusabilidade*, que establece a reusabilidade dos modelos xenéricos ou parciais previamente existentes na creación de novos modelos.
6. *Principio de separación do comportamento e a funcionalidade*, que establece a distinción entre as funcións a realizar e como son realizadas, para garantir a flexibilidade organizativa da empresa.
7. *Principio de separación de proceso e recurso*, que establece a distinción entre as operacións realizadas e as entidades que as realizan, para garantir a flexibilidade operativa da empresa.
8. *Principio de conformidade*, que establece que as linguaxes de modelado deben ser consistentes e non redundantes.
9. *Principio de visualización do modelo*, que establece que as linguaxes de modelado deben ter representacións gráficas simples e non ambiguas.

En [124] identifícanse como básicos os principios de xenericidade, modularidade e reusabilidade na aplicación dunha aproximación ao desenvolvemento de sistemas industriais mediante a utilización de bloques de modelado predefinidos e librarías de módulos "software" de diferentes provedores. As ferramentas informáticas que proporcionen soporte a esta aproximación deben cumprir os seguintes requisitos [169]:

1. Proporcionar bloques de modelado xenéricos, representados mediante "templates" e implementados mediante técnicas de orientación a obxectos.
2. Proporcionar librarías de modelos (parciais e particulares) que podan ser reutilizados.
3. Permitir o acceso aos modelos dende diferentes puntos de vista, como mínimo o funcional, de información, de recursos e o organizativo comentados anteriormente.
4. Proporcionar soporte nas diferentes fases do ciclo de vida do sistema comentados anteriormente: requirimentos, deseño e implementación.

Considerase en [169] que unha plataforma formada por unha interface gráfica sobre un ambiente de programación orientado a obxectos é adecuada para a implementación destas ferramentas, e que a principal dificultade está en proporcionar ambientes gráficos amigábeis para o usuario ao tempo que se oculta a complexidade do manexo dos modelos e se proporcionan funcións eficaces de análise e simulación.

2.2.3. Características do "software" de control industrial

O "software" de control industrial ten características coincidentes coas do "software" para sistemas de tempo real, aínda que tamén presenta algunhas peculiaridades. Sen ánimo de ser

exhaustivos poden considerarse as características indicadas a continuación como as máis significativas das aplicacións de control industrial:

1. Están deseñadas para interactuar directa ou indirectamente cun proceso físico, xa sexa en tarefas de control, supervisión ou coordinación, polo que a consideración dos requisitos temporais do proceso é un aspecto fundamental no deseño e implementación da aplicación para garantir a reactividade e a seguridade na operación do sistema de control.
2. Teñen estructuras "software" complexas organizadas xerarquicamente dende un punto de vista lóxico e que poden reconfigurarse dinamicamente para adaptarse a diferentes modos de operación. Fisicamente están distribuídas en sistemas como o da Figura 2.3, nos que poden realizarse múltiples actividades simultáneas en cada nodo. En consecuencia a estructuración xerárquica, a configuración dinámica, a distribución e a concorrencia son características inherentes deste tipo de aplicacións.
3. Integran múltiples descricións que utilizan distintos puntos de vista do sistema: a estructura física e o comportamento dinámico do proceso; a estructura física, a funcional, os modos de operación e o comportamento dinámico do sistema de control; a información utilizada, etc. Incorporan polo tanto diferentes formalismos de modelado para representar os aspectos estáticos, dinámicos, funcionais e informacionais da aplicación.
4. Integran modelos da dinámica do proceso, da dinámica do sistema de control ou dambas. Estes modelos son representados utilizando formalismos que permiten verificar durante o deseño as propiedades formais do modelo mediante técnicas analíticas ou por simulación.

2.2.4. Aplicación de metodoloxías "software" en sistemas industriais

Dadas as características descritas anteriormente, pode deducirse que o desenvolvemento de "software" en sistemas industriais require de metodoloxías que combinen unha aproximación "software" (con características de tempo real) con formalismos para o modelado dinámico de sistemas [181]. A utilización conxunta dambas aproximacións permite combinar as capacidades de modelado propias das metodoloxías "software" (diferentes niveis de abstracción, distintos puntos de vista, etc.) coas que proporcionan os formalismos de modelado dinámico de sistemas (verificación formal, simulación, xeración automática de código, etc.). As diferentes propostas realizadas neste sentido poden clasificarse en catro categorías:

1. A inclusión dalgún formalismo de modelado dinámico de sistemas como parte da metodoloxía de desenvolvemento "software", definindo as regras de integración do novo formalismo cos demais modelos da metodoloxía. Algúns exemplos son o Grafcet con OMT [157] ou as RdP con OMT [12].
2. A definición de semánticas formais para os modelos 'informais' da metodoloxía "software". Algúns exemplos son os diagramas de actividades en SADT [145] ou os StateCharts en OMT [80].
3. A asociación dunha semántica formal aos modelos 'informais' da metodoloxía "software" mediante a especificación dun algoritmo que permita convertelos a unha representación na que se utilice un formalismo de modelado dinámico de sistemas. Algúns exemplos son SADT e RdP [181], SA/RT e RdP [57] ou UML e RdP [23].
4. A proposta de novos formalismos que incorporan capacidades adicionais (estructuración xerárquica, abstracción, reusabilidade) aos formalismos de modelado dinámico de sistemas básicos. Algúns exemplos son os Object-StateCharts [180] ou as RdP de alto nivel ("High-Level Petri Nets", [91]).

A aproximación utilizada nesta tese de doutoramento pertence á primeira das categorías indicadas anteriormente, propoñendo a utilización conxunta do Grafcet coa UML [29] definindo a relación entre ambos a través dos seus metamodelos.

2.3. A ferramenta proposta

Neste apartado descríbense as avantaxes proporcionadas no desenvolvemento de "software" para sistemas industriais polas características implementadas na ferramenta proposta nesta tese de doutoramento: a orientación a obxectos e o modelado de DEDS mediante Grafcet. Ademais compárase a aproximación utilizada con outras aproximacións relacionadas; descríbese a arquitectura lóxica da ferramenta e o proceso de desenvolvemento dunha aplicación con ela; e indícanse as técnicas dispoñíbeis para a integración con outras aplicacións. En conxunto a ferramenta implementada proporciona as funcionalidades precisas para obter automaticamente, a partir dos modelos especificados nas fases de análise e deseño, unha versión da aplicación executábel nunha máquina virtual deseñada para dar soporte a aplicacións dirixidas por eventos en ambientes distribuídos como o da Figura 2.3.

2.3.1. A orientación a obxectos no modelado de sistemas industriais

No modelado de sistemas industriais téñense aplicado diferentes aproximacións como a funcional (SADT, IDEF), a de modelado da información (diagramas entidade-relación [41]) ou a da especificación da dinámica (RdP). A utilización da aproximación orientada a obxectos está obtendo cada vez maior aceptación debido a que os principios nos que se sustenta (capsulación, abstracción, modularidade e xerarquización) proporcionan múltiples avantaxes no modelado e simulación de sistemas complexos heteroxéneos:

1. *O soporte á reusabilidade* dos modelos, mediante mecanismos como a herdanza ou a composición.
2. *A modularidade*, que permite capsular os modelos para formar compoñentes con interfaces ben definidas que poden organizarse en librarías para ser reutilizados.
3. *A flexibilidade* dos modelos orientados a obxectos, que poden ser modificados e ampliados engadindo, eliminando ou modificando obxectos e clases.
4. *A portabilidade* dos modelos, que poden ser compartidos entre diferentes plataformas.
5. *A expresividade* da orientación a obxectos, que utiliza un concepto uniforme en diferentes niveis de detalle e fases do ciclo de vida do sistema. ademais poden obterse versións executábeis dos modelos de maneira natural utilizando unha linguaxe de programación orientada a obxecto.
6. *A consistencia* entre os distintos puntos de vista do modelo.
7. A dispoñibilidade dunha linguaxe unificada de modelado (UML [29]) e dunha infraestructura de execución distribuída (CORBA [126]) estandarizadas.
8. A dispoñibilidade de multitude de metodoloxías, métodos, linguaxes e ferramentas para a análise, deseño, simulación e programación orientada a obxectos.

Como exemplos da aplicación da aproximación orientada a obxectos en sistemas industriais poden citarse os seguintes:

- Entre as metodoloxías: IEM [117], metodoloxía que combina conceptos da orientación a obxectos con diagramas de actividades baseados nos de IDEF; CIMOSA [4], que se ben en

si mesma non é unha metodoloxía orientada a obxectos, si se identifica esta aproximación como válida para a representación e implementación de parte das propostas que nela se fan.

- Entre as ferramentas: MOSES [112], un ambiente para o modelado de sistemas híbridos complexos; FBCad [111], un ambiente de programación baseada en compoñentes para o desenvolvemento de bloques función reutilizabeis que soporta o deseño, simulación e xeración de código para sistemas distribuídos; G2 [68], un ambiente de programación de sistemas de supervisión 'intelixentes' que combina unha aproximación orientada a obxecto con mecanismos de busca heurística baseados en regras.

Un dos principais focos de interese da aplicación da aproximación a obxectos a sistemas industriais é que soporta de forma natural a construcción de librarías de bloques xenéricos para o modelado de sistemas que, ao ser representados mediante "templates", poden ser reutilizados ou adaptados para obter modelos particulares, utilizando os mecanismos de instanciación, herdanza ou composición propios da orientación a obxectos. A simulación e xeración do código dos modelos particulares é directa utilizando o soporte proporcionado polas ferramentas e linguaxes dispoñíbeis. A principal carencia é a non dispoñibilidade na industria dun modelo de referencia estandarizado, unha proposta neste sentido pode consultarse en [36].

2.3.2. O Grafcet como formalismo de especificación

Dacordo a [99], os requisitos que debe cumprir un formalismo para a especificación de sistemas de control complexos son:

1. Debe ser gráfico e fácil de utilizar polo enxeñeiro de control.
2. Debe estar baseado nun método xerárquico que permita diferentes niveis de abstracción.
3. Debe basearse nunha aproximación híbrida que integre un modelo discreto para a coordinación da lóxica de execución das actividades de control e un modelo continuo para a síntese das leis de control.
4. Debe permitir a representación de comportamentos xenéricos que representen invariantes en certos dominios de aplicación que permitan derivar modelos particulares para os sistemas de control pertencentes a eses dominios.

Analizando o Grafcet dende o punto de vista dos criterios anteriores, as conclusións son as seguintes:

1. O Grafcet é un formalismo gráfico estandarizado [84][85] para a especificación de controladores lóxicos secuenciais, fácil de utilizar, amplamente difundido e independente da tecnoloxía de implementación utilizada.
2. O Grafcet define unha estructura xerárquica (§3.2.2.3) e permite a utilización de diferentes niveis de detalle en accións e condicións, polo que pode ser utilizado con métodos nos que se aplique unha estratexia baseada no refinamento sucesivo a diferentes niveis de abstracción.
3. O Grafcet é especialmente adecuado para a coordinación de diferentes actividades. En [30] descríbense tres posíbeis formas de utilizar o Grafcet na estructuración dun sistema de control:
   a. A estructura global do sistema modelase usando algún formalismo distinto ao Grafcet, e os módulos que a forman son especificados utilizando distintos formalismos, incluído o Grafcet (Figura 2.4.a). Os módulos que utilicen Grafcet interaccionan cos demais mediante o intercambio de valores booleanos.

b. A estructura global do sistema modélase utilizando o Grafcet, e os módulos que a forman son especificados utilizando distintos formalismos, incluído o Grafcet (Figura 2.4.b). A coordinación entre a estructura global e os módulos que utilicen outros formalismos realízase mediante o intercambio de valores booleanos.

c. A estructura e os módulos que a forman son especificados utilizando o Grafcet (Figura 2.4.c).

Estas estructuracións están baseadas na definición orixinal do Grafcet [84], que estaba dirixida á especificación de controladores lóxicos nos que as variábeis utilizadas eran unicamente de tipo booleano. Recentemente a revisión do estándar [85] adoptou as propostas de [75] e [76] para ampliar a aplicabilidade do Grafcet a sistemas híbridos. Aplicacións do Grafcet na estructuración do control de sistemas híbridos e "batch" son descritas en [172] e [95], respectivamente.

4. O estándar Grafcet non inclúe un medio de representar comportamentos xenéricos a partir dos que poidan derivarse modelos particulares. Existen propostas baseadas en aproximacións orientadas a obxecto que inclúen esta capacidade: parametrización de grafcets, macroetapas e procedementos en [94] ou herdanza e agregación de modelos dinámicos en [157].

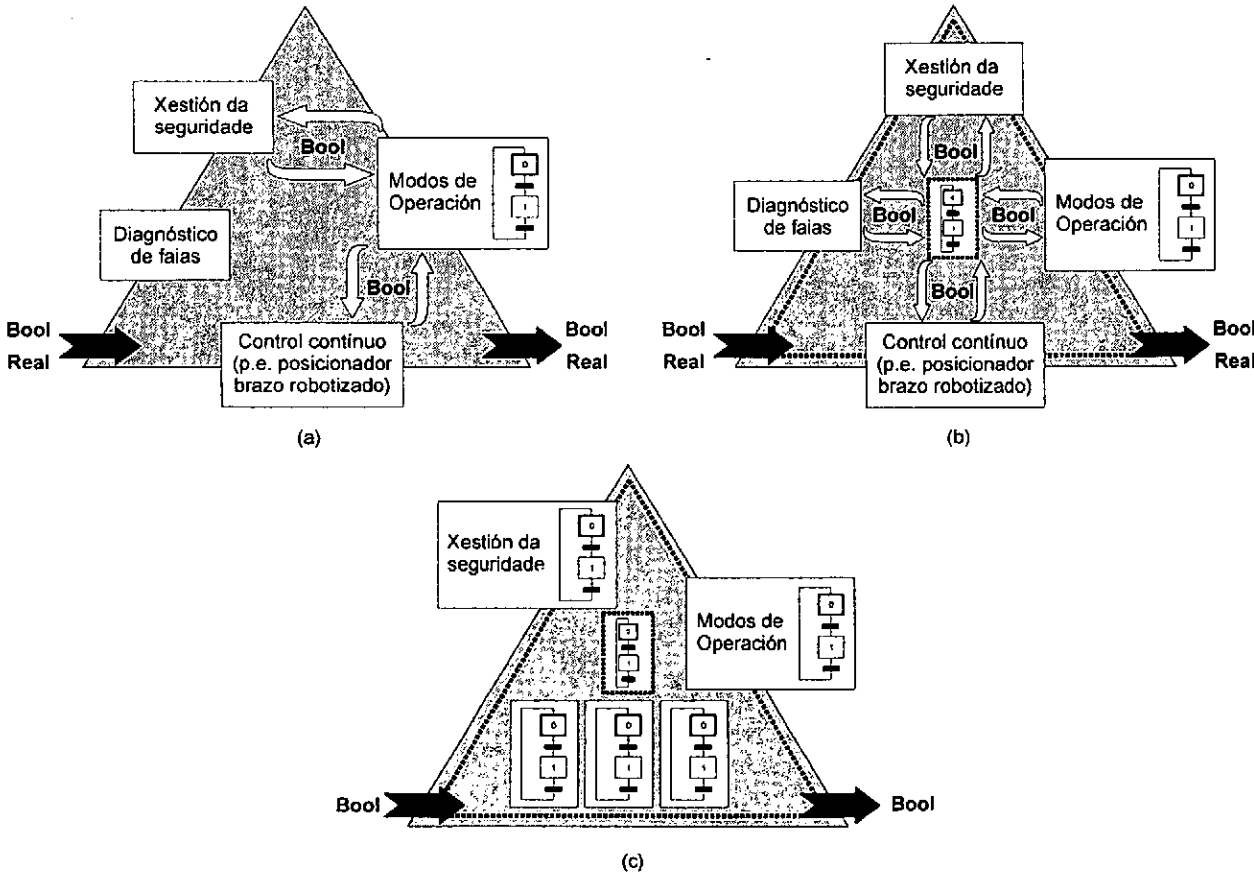


Figura 2.4. Formas de estructurar un sistema de control utilizando Grafcet: a) o Grafcet é utilizado nalgún dos módulos que forman o sistema de control; b) o Grafcet é utilizado para a coordinación dos módulos do sistema de control; e c) o sistema de control é especificado completamente utilizando o Grafcet.

Ademais das características anteriores, o Grafcet ofrece outras avantaxes:

1. Ao ser un formalismo estandarizado e gráfico, facilita a comunicación entre os diferentes técnicos e futuros usuarios implicados no proceso de desenvolvemento do sistema de control.
2. Mediante o soporte informático adecuado, pode utilizarse o mesmo formalismo dende a especificación do sistema ate a súa operación e mantemento [30]. O propio formalismo pode utilizarse como interface gráfica para a simulación e monitorización do sistema.
3. O Grafcet pode utilizarse para estructurar o sistema dende diferentes puntos de vista [95]: funcional, físico, modos de operación, etc.
4. O Grafcet permite automatizar diferentes funcións de control utilizando o mesmo formalismo: coordinación, seguridade, detección de fallas [62][63], supervisión [61], etc.
5. O Grafcet é un formalismo de especificación independente da tecnoloxía utilizada na súa implementación (p.e. pneumática, electromecánica, computador) e da distribución física dos modelos.
6. Disponse de diferentes métodos formais para a validación, verificación e síntese de modelos Grafcet (§3.1) e dunha versión estandarizada [86] adaptada á programación de PLCs —o SFC (§3.6.1)—.

2.3.3. Comparación con outras aproximacións

Descríbense a continuación as principais características que distinguen o traballo realizado nesta tese de doutoramento doutros relacionados. Nótese que son moitas as aproximacións existentes para o desenvolvemento de "software" en sistemas de control industrial, polo que unicamente se citan aquelas que se consideraron conceptualmente máis próximas á proposta realizada:

1. Utilización do Grafcet como parte dunha metodoloxía para o desenvolvemento de sistemas de control industrial. Existen varias propostas de metodoloxías que utilizan o Grafcet para a especificación das secuencias lóxicas das actividades de control identificadas por medio da aplicación dalgún método, xa sexa de descomposición funcional ou orientada a obxectos. O Grafcet utilízase unicamente a nivel local e a coordinación de actividades modélase mediante outro formalismo DEDS (RdP, StateCharts). Como exemplos pódense citar: SADT con RdP [181], OMT con extensións para o modelado de xerarquías funcionais [181] ou uso de diagramas de fluxo e RdP [37].

   Con respecto a estas metodoloxías a ferramenta proposta é complementaria. Utilizando o metamodelo Grafcet como formato de representación dos modelos poden integrarse as ferramentas que dean soporte a estas metodoloxías co compilador e o intérprete Grafcet, tal e como se explica en (§2.3.6). Isto posibilita o paso automático da especificación á implementación e a dispoñibilidade dun ambiente para a simulación e operación dos modelos.
2. Integración do Grafcet nunha metodoloxía "software" como formalismo alternativo para a especificación do comportamento. O único traballo neste sentido do que o autor desta tese ten noticia é a proposta realizada en [157]. Nese traballo proponse un metamodelo para integrar o Grafcet na metodoloxía OMT e enúncianse os principios para a reutilización dos modelos mediante os mecanismos de herdanza e agregación. Este é un traballo inicial que enuncia unhas ideas básicas dende o punto de vista do deseño. Non se tratan aspectos importantes como a semántica e a implementación dos modelos, e o metamodelo proposto non inclúe todas as características comentadas en (§3.2.2).

3. Utilización dun formalismo baseado no Grafcet: o GraphChart [9]. Este formalismo basease na sintaxe e semántica do Grafcet e inclúe conceptos avanzados como a parametrización, o uso de métodos ou o paso de mensaxes, tomados da aproximación orientada a obxectos e das RdP de alto nivel [91]. O GraphChart foi aplicado na especificación de supervisores intelixentes [9] e no modelado de receitas en sistemas "batch" [95]. En [94] descríbense as diferencias entre o Grafcet e o GraphChart. O principal inconveniente do GraphChart dende o punto de vista práctico é que só se dispón dunha implementación en G2 [68], un ambiente orientado ao desenvolvemento de aplicacións de supervisión intelixente. Isto limita a aplicabilidade do formalismo e require de soporte adicional para a xeración de código eficiente. Actualmente está en proxecto unha versión Java, denominada JGraphChart, que aínda non estaba dispoñíbel no momento de redactar esta tese.
4. Aproximacións baseadas nas RdP. Existen numerosos traballos que utilizan as RdP como formalismo para o desenvolvemento de "software" de control. Entre os máis relacionadas cos obxectivos desta tese poden citarse: a proposta dun método baseado na lóxica de paso de testigo para a obtención dunha representación en linguaxe de contactos dunha RdP dada [96]; a integración das RdP nunha ferramenta de desenvolvemento "software" baseada na utilización de UML [47]; a proposta dunha metodoloxía de desenvolvemento "software" para a obtención de programas en lista de instruccións (IL) para PLCs a partir de modelos formais que utilizan un tipo de RdP [98]; e a proposta dun metamodelo para as RdP que utiliza conceptos de modelado orientado a obxectos e que pode ser utilizado no deseño de compoñentes "software" xenéricos orientados ao control industrial [109].

   O Grafcet proporciona capacidades de modelado, análise e verificación próximas ás das RdP (§3.4.4.1), e é ademais un formalismo estandarizado, fácil de aprender e interpretar, e unha linguaxe estándar de programación de PLCs baseada nos seus conceptos. O Grafcet é mellor aceptado nos ambientes industriais sobre todo durante a operación do sistema, xa que os modelos obtidos coas RdP tenden a ser complexos e difíciles de interpretar. O ámbito de aplicabilidade da ferramenta proposta está máis orientado ás fases de deseño, implementación e operación da aplicación que ás de análise e verificación, polo que o Grafcet foi o formalismo escollido. É factíbel tamén a utilización de aproximacións híbridas que utilicen ambos formalismos, integrando a ferramenta proposta con algunha outra que dea soporte ás RdP mediante os mecanismos descritos en (§2.3.6).
5. Aproximacións baseadas nos StateCharts. Os StateCharts foron inicialmente propostos para a especificación de sistemas reactivos complexos descritos mediante diagramas de estados e transicións aos que basicamente se lles engade a concorrencia e unha estructura xerárquica. Foron incluídos como formalismos para a especificación do comportamento en metodoloxías orientadas a obxectos como Booch ou UML, e son utilizados con pequenas variacións en ferramentas comerciais como o StateFlow de MatLab, o Statemate[21], ou o anyStates[22]. O Grafcet proporciona capacidades de modelado próximas ás dos StateCharts (§3.4.4.3), e as razóns para á súa utilización son semellantes ás comentadas para as RdP. A utilización de aproximación híbridas que combinen ambos formalismos facilita a aplicación de ferramentas de desenvolvemento "software" xenéricas na implementación de controladores industriais. Como se comentou anteriormente, nesta tese proponse con ese fin un metamodelo para o Grafcet integrado no de UML (§5.1). Outra aproximación que

[21] Statemate® é un producto da compañía I-Logix. http://www.ilogix.com/
[22] AnyStates® é un producto da compañía XJ Technologies. http://www.xjtek.com/

propón a utilización do SFC para a descrición detallada das accións estáticas nos estados dun StateChart pode consultarse en [19].

6. Ambientes de programación SFC/Grafcet. Existe unha variedade de ferramentas dispoñíbeis na actualidade, orientadas sobre todo á programación de PLC's ou "softPLCs", que inclúen o SFC ou algunha versión máis ou menos completa do Grafcet como linguaxe gráfica de programación. Algunhas destas aplicacións son analizadas en detalle no Capítulo 4. As principais aportacións da ferramenta proposta, considerando as conclusións extraídas da análise realizada nese capítulo, son as seguintes:
   a. Defínese un metamodelo para o Grafcet relacionado co da UML que pode ser utilizado como formato de intercambio dos modelos Grafcet entre aplicacións ou para integrar o Grafcet en ambientes CASE baseados en UML e utilizalo así como alternativa para a especificación do comportamento das compoñentes dinámicas dos modelos.
   b. As funcionalidades para a representación, compilación, simulación e execución de modelos Grafcet poden ser utilizadas de maneira integrada dende o editor gráfico da ferramenta ou de maneira independente dende outras ferramentas que utilicen os mecanismos de integración indicados en (§2.3.6).
   c. A interpretación de modelos pode ser configurada para elixir o algoritmo a utilizar e a forma en que os eventos, variábeis e accións serán considerados durante as evolucións internas utilizando interpretacións semánticas ben definidas (§3.3.2).
   d. Disponse dun ambiente de execución (a máquina virtual) flexíbel, portábel, escalábel e configurábel dinamicamente, que facilita a aplicabilidade en ambientes distribuídos e heteroxéneos. Ademais a súa funcionalidade non se limita á interpretación de modelos Grafcet, senón que proporciona os mecanismos básicos (xestión de eventos, temporizacións e variábeis de E/S) que permitirán integrar intérpretes doutros formalismos DEDS en futuras versións.
   e. A linguaxe de programación de accións utilizada é o C++, o que posibilita a utilización do grande número de librarías externas existentes e que, conxuntamente co meetamodelo Grafcet definido, permiten integrar o Grafcet cunha aproximación orientada a obxecto no desenvolvimento de "software" para sistemas de control industrial.

2.3.4. A arquitectura da ferramenta

A arquitectura lóxica da ferramenta proposta está formada por dous subsistemas: o subsistema de desenvolvemento (Figura 2.5.a) e o de execución (Figura 2.5.c). O subsistema de desenvolvemento está formado por un editor que proporciona unha interface gráfica a través da que o usuario accede ás funcionalidades da ferramenta e un compilador que xera o código a executar no subsistema de execución. O subsistema de execución está composto por unha máquina virtual que contén un intérprete Grafcet e os módulos que permiten a interacción da máquina virtual cun proceso físico e cos servizos proporcionados polo "hardware" e o sistema operativo (procesos, temporizacións) no que se execute. Ambos subsistemas interaccionan a través dos servizos de acceso remoto da máquina virtual que permiten a consulta e modificación da súa configuración e a carga, descarga e control da execución das aplicacións.

Os subsistemas poden executarse en equipos diferentes ou no mesmo equipo (Figura 2.5.b), que é a configuración habitual utilizada durante a simulación. Esta arquitectura é modular e facilmente escalábel, pudendo utilizarse configuracións distribuídas nas que unha ou máis máquinas virtuais se utilicen conxuntamente cun ou máis subsistemas de desenvolvemento. Sen embargo para que a ferramenta sexa plenamente operativa en ambientes distribuídos é preciso

incluir certas funcionalidades que non foron consideradas na versión actual, como por exemplo a edición simultánea de modelos, asistencia á compilación e distribución automática de aplicacións en redes heteroxéneas, ou o intercambio de información de estado entre máquinas virtuais.

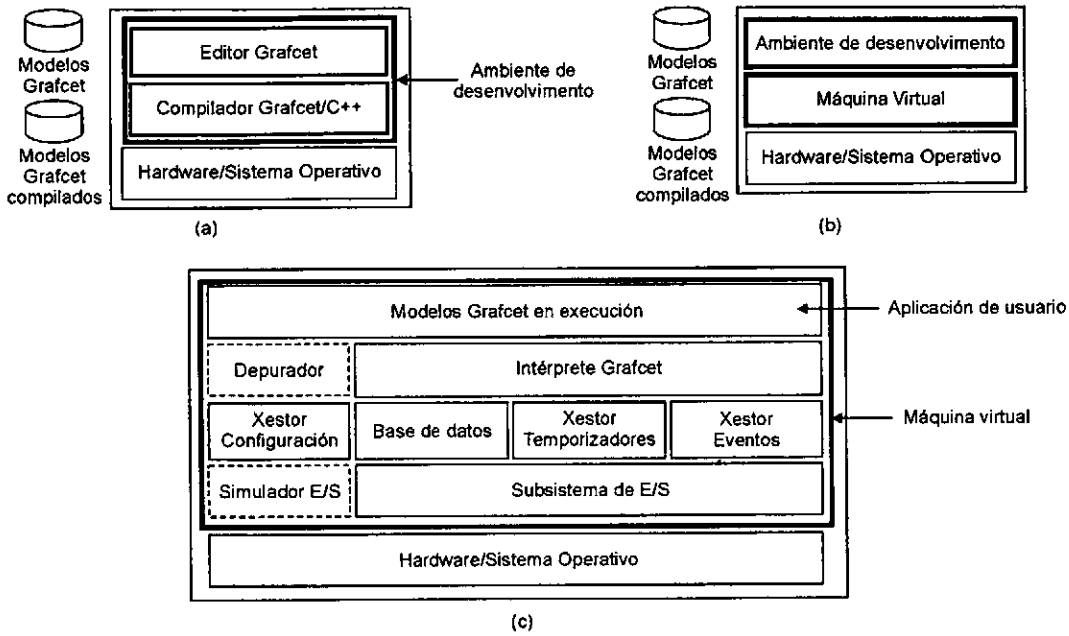


Figura 2.5. Arquitectura da ferramenta proposta: a) subsistema de desenvolvemento; b) configuración co subsistema de desenvolvemento e execución no mesmo equipo; e c) subsistema de execución.

2.3.5. O proceso de desenvolvemento coa ferramenta

A Figura 2.6 mostra un diagrama que representa o proceso de desenvolvemento dunha aplicación utilizando as compoñentes da ferramenta implementada. O editor Grafcet proporciona unha interface gráfica común a través da que o usuario accede ás funcionalidades da ferramenta, sen embargo, tanto o compilador como a máquina virtual non dependen do editor gráfico e poden ser utilizadas por separado como aplicacións autónomas.

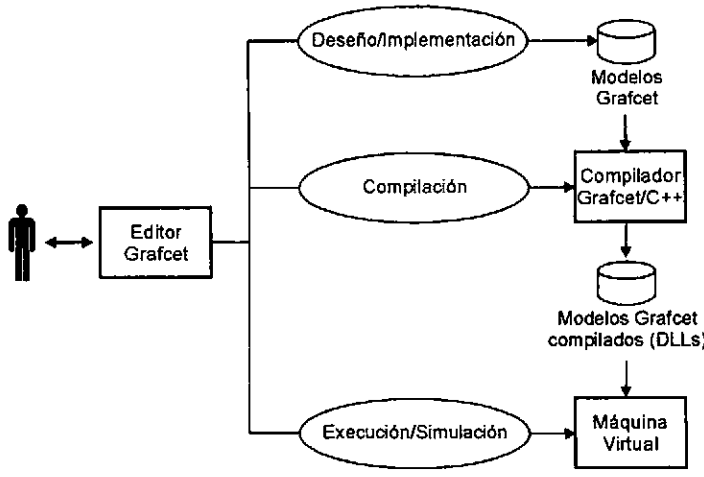


Figura 2.6. Proceso de desenvolvemento dunha aplicación coa ferramenta implementada.

No editor gráfico o enxeñeiro utiliza unha combinación de C++ e Grafcet para modelar a estructura do sistema de control a diferentes niveis de abstración e dende diferentes puntos de vista. O código C++ utilizado para estructurar os modelos pode codificarse manualmente ou importarse o xerado automaticamente cunha ferramenta CASE externa. Os métodos definidos nas interfaces públicas das clases que compoñen os modelos poden ser utilizados dende o código de accións e condicións, que tamén será programado en C++. O Grafcet é utilizado no editor para a especificación das secuencias de estados de operación (§2.1.1) das compoñentes identificadas nos modelos, así como para a descrición detallada da lóxica secuencial deses estados. As colaboracións entre compoñentes e a dinámica global do sistema de control tamén son modeladas mediante Grafcet.

O editor proporciona asistencia durante a edición gráfica dos modelos Grafcet, facilitando a súa estructuración, a súa verificación sintáctica, a codificación de accións e condicións e a declaración das variábeis utilizadas. Unha vez finalizada a edición obtense unha versión executábel do modelo mediante o compilador Grafcet. Este converte o modelo Grafcet a unha representación C++ equivalente e utiliza un compilador C++ externo para obter unha versión compilada en forma de DLL. O compilador C++ utilizado pode ser un compilador nativo ou cruzado, dependendo de se o sistema operativo do equipo no que se executa a máquina virtual coincide ou non co do equipo no que se executa o subsistema de desenvolvemento.

A versión executábel do modelo pode utilizarse tanto para a validación do sistema (simulando as E/S) como para o control do proceso. A máquina virtual pode ser configurada para axustar os tempos de resposta do sistema dependendo dos requisitos temporais do proceso. Ademais a máquina virtual implementa un modo de depuración que permite visualizar no editor gráfico o estado de evolución do modelo Grafcet e que pode ser utilizado dende aplicacións externas para a implementación dunha interface gráfica de operador que permita a monitorización e a supervisión do proceso durante a súa operación.

Nótese que o proceso descrito anteriormente pode repetirse múltiples veces durante o desenvolvemento dunha aplicación, xa sexa en fases diferentes (p.e. análise, deseño, implementación, probas, operación, mantemento); con distintos niveis de refinamento (p.e. especificación inicial, especificación detallada, implementación); ou concentrándose en partes diferentes dun mesmo modelo. A ferramenta utilízase de maneira uniforme nas diferentes fases e niveis de abstracción, o que a fai especialmente indicada para procesos de desenvolvemento de "software" iterativos baseados na construcción e refinamento progresivo de modelos gráficos executábeis.

2.3.6. Integración da ferramenta con outras aplicacións

A ferramenta proposta foi deseñada para ser utilizada en ambientes de desenvolvemento de "software" de control heteroxéneos, nos que se utilice conxuntamente con outras aplicacións para o modelado de sistemas continuos, o deseño orientado a obxectos, o modelado de DEDS utilizando formalismos distintos a Grafcet como os StateCharts ou as RdP, etc. A ferramenta proporciona tres formas diferentes de integración con outras aplicacións:

1. A inclusión dos modelos desenvolvidos noutras ferramentas como parte do modelo Grafcet a executar na máquina virtual, que pode facerse de dúas maneiras: obtendo unha versión C++ dos modelos e importándoa no editor (Figura 2.7.a); ou obtendo unha versión compilada que poida enlazarse á DLL da aplicación (Figura 2.7.b).
2. A utilización do mecanismo de simulación da máquina virtual para o intercambio de información en tempo de execución (§7.2.3). Neste caso a máquina virtual executa

unicamente o modelo Grafcet, e os modelos definidos coas outras ferramentas son executados en aplicacións externas á máquina virtual (Figura 7.17).

3. Utilizando o metamodelo Grafcet proposto como parte da ferramenta e a libraría C++ que o implementa (§5.2). Tanto o editor Grafcet como o compilador traballan con modelos Grafcet representados utilizando este metamodelo, que pode ser utilizado basicamente de dúas maneiras (Figura 2.7.c):
   a. Como forma de integración do Grafcet como alternativa aos StateCharts en ferramentas de deseño orientado a obxecto baseadas en UML.
   b. Como formato de intercambio coas aplicacións externas utilizadas para a verificación formal e a análise de modelos Grafcet.

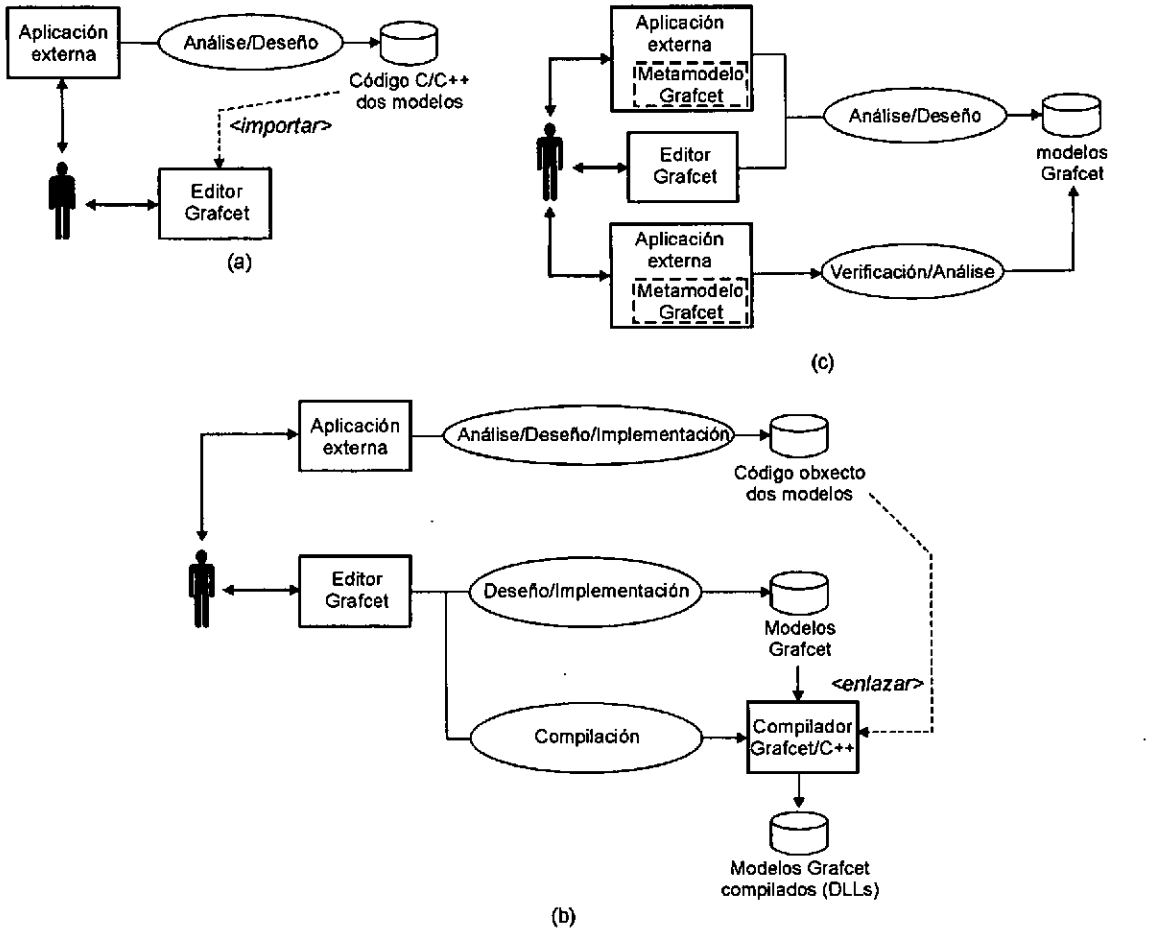


Figura 2.7. Formas de integración da ferramenta con outras aplicacións: a) mediante código fonte C++; b) mediante módulos compilados; e c) mediante modelos baseados no metamodelo Grafcet.

2.4. Conclusións

Neste capítulo explicáronse as funcionalidades básicas da ferramenta proposta nesta tese de doutoramento introducíndose a súa arquitectura lóxica, a súa utilización no proceso de desenvolvemento dunha aplicación e os mecanismos de integración con outras aplicacións nun ambiente heteroxéneo de desenvolvemento de "software" de control industrial. As características da orientación a obxectos e o modelado de sistemas dinámicos de eventos discretos foron situadas en contexto dentro do proceso máis xenérico da enxeñería de sistemas industriais utilizando unha aproximación baseada na dispoñibilidade de modelos reutilizabeis.

As compoñentes da ferramenta proposta proporcionan en conxunto as funcionalidades precisas que permitan a un enxeñeiro de control obter automaticamente o código dunha aplicación executábel nun ambiente industrial distribuído a partir dos modelos especificados nas fases de análise e deseño, combinando unha aproximación orientada a obxecto cun formalismo gráfico para a especificación de controladores lóxicos secuenciais.

Como se explica nos capítulos posteriores, no deseño e implementación da ferramenta tomáronse como prioridades a modularidade, portabilidade e aplicabilidade, evitando no posíbel as dependencias de sistemas ou tecnoloxías específicas. A única excepción é a do editor gráfico, xa que as librarías portábeis existentes non permitían as mesmas posibilidades que as específicas dun sistema concreto, por ese motivo decidiuse a utilización dunha libraría dependente do sistema operativo Windows. O resto de compoñentes foron implementadas na súa versión inicial para traballar no sistema operativo Windows utilizando unha rede de comunicacións TCP/IP, aínda que a súa portabilidade a outros sistemas operativos e redes de comunicacións está garantida mediante a utilización de interfaces abstractas no deseño da aplicación que permiten eliminar as dependencias dos aspectos específicos de cada sistema. O soporte dun novo sistema require unicamente a implementación desas interfaces abstractas utilizando as funcionalidades específicas que este proporcione.

O traballo realizado nesta tese de doutoramento permitiu obter unha versión inicial da ferramenta proposta que soporta as características indicadas neste capítulo. Sen embargo existen algúns aspectos que deben ser revisados en futuras versións para mellorar a súa aplicabilidade, principalmente no referente ao soporte á distribución de aplicacións, a definición de mecanismos de reutilización de modelos Grafcet compatíbeis cos de herdanza e agregación da aproximación orientada a obxectos, ou o soporte dalgunha linguaxe de descrición de dispositivos de E/S nos "drivers" da máquina virtual, por exemplo.

Capítulo 3. O Grafcet

3.1. Introducción

O Grafcet é un formalismo gráfico baseado nas RdP definido co propósito de dispor dun medio normalizado de descrición de controladores lóxicos que fora independente da tecnoloxía utilizada na súa implementación. Foi inicialmente proposto pola AFCET [1], unha asociación francesa formada a partes iguais por membros de universidades e empresas. Posteriormente foi adoptado como estándar, primeiro en Francia [164] e con posterioridade a nivel internacional [84]. O seu uso espallouse na industria europea e na educación [97], e a súa introducción no mercado americano foi medrando progresivamente [15][16]. Coa adopción no estándar IEC 61131-3 [86] do SFC, unha versión lixeiramente modificada do Grafcet, como linguaxe gráfica para a programación de PLCs (§3.6), o Grafcet converteuse nun formalismo amplamente recoñecido e utilizado. Entre as avantaxes que proporciona poden citarse:

1. É un formalismo normalizado, gráfico, fácil de entender e utilizar que facilita a comunicación entre os técnicos implicados no desenvolvemento dun sistema de control.
2. É independente da tecnoloxía de implementación utilizada polo que permite a análise e especificación do sistema antes de pasar a considerar os aspectos relacionados coa implementación.
3. Pode ser aplicado en diferentes fases e a diferentes niveis de detalle da especificación, polo que pode utilizarse con métodos de deseño baseadas no refinamento sucesivo da solución.
4. Propón unha interpretación única das E/S como valores booleanos o que facilita a expresión das relacións lóxicas que definen o comportamento do controlador.
5. A súa definición está baseada na das RdP, polo que se dispón dunha base matemática que pode ser utilizada en métodos de análise e verificación formal.
6. Disponse dunha versión estandarizada para a programación de PLCs, polo que o paso dende a especificación á implementación en ambientes de programación IEC é case directo.

No que respecta aos aspectos negativos, o Grafcet ten sido criticado por non dispor dunha semántica formal que permita garantir os requisitos de corrección e seguridade, e tamén por non existir unha metodoloxía que permita utilizalo no desenvolvemento eficiente de modelos de calidade cando se traballa con sistemas complexos. Co ánimo de superar estas limitacións desenvolvéronse (principalmente en Francia) numerosos traballos de investigación orientados basicamente en catro direccións [182]:

1. **Extensións sintácticas e semánticas**

As extensións sintácticas propostas permiten estructurar xerarquicamente os modelos Grafcet [103], facilitan o manexo de diferentes niveis de abstracción [55][73] e melloran a representación de sistemas híbridos [75][76]. No referente ás extensións semánticas propuxéronse interpretacións que garanten a sincronía, determinismo e reactividade dos modelos. Parte destas propostas estandarizaronse en [165] e son explicadas en (§3.2.2; §3.3.2).

2. **Metodoloxías**

Nesta liña realizáronse algúns traballos que propoñen metodoloxías orientadas ao desenvolvemento de sistemas automatizados de fabricación nas que se utiliza o Grafcet e que cobren o ciclo de vida do sistema. A maioría destes traballos parten dunha metodoloxía de desenvolvemento "software" á que lle incorporan algún formalismo de modelado de DEDS e definen as regras que permiten converter os modelos representados mediante ese formalismo a especificacións de implementación utilizando Grafcet. Algunha destas propostas utilizan unha aproximación funcional [37][120][163] e outras unha aproximación orientada a obxecto [173][181].

3. **Verificación e validación das especificacións**

A verificación de determinadas propiedades dos modelos Grafcet como a consistencia, estabilidade, capacidade de reiniciación, ausencia de bloqueo, etc. e a validación das súas propiedades temporais, restriccións de seguridade, etc. son campos nos que se produciu na pasada década unha grande actividade investigadora. Entre o grande número de técnicas propostas pode diferenciarse entre as que teñen en conta unicamente o modelo Grafcet e as que teñen en conta tamén o comportamento do sistema a controlar. Entre as primeiras hai propostas baseadas no uso de linguaxes síncronas [7][115], autómatas con información temporal [116], RdP [13], álxebra Max+ [129], máquinas de estados [38][148] ou autómatas híbridos [79]. E entre as segundas hainas baseadas no uso de TCCS/TML [13] e no de TTM/RTTL [48].

4. **Síntese dos modelos**

As investigacións nesta liña baséanse na teoría do control supervisor [177] e teñen como obxectivo obter métodos para a síntese do grafcet supremo [40][183][184], que é o que se obtén ao aplicar as mínimas restriccións posíbeis ao comportamento dun grafcet dado para cumprir unhas determinadas especificacións de seguridade e garantir a ausencia de bloqueo durante o funcionamento do sistema de control.

Recentemente aprobouse unha revisión do estándar Grafcet internacional [85] que inclúe algunha das propostas anteriores. O traballo realizado nesta tese de doutoramento está baseado nas definicións anteriores a esta revisión. Para unha discusión sobre como poden representarse algunhas das extensións adoptadas nesa revisión na ferramenta proposta nesta tese poden consultarse [133] e [134].

O resto deste capítulo estructúrase da maneira seguinte: o apartado (§3.2) dedícase á sintaxe do Grafcet, descríbense neste apartado os elementos sintácticos básicos e as extensións propostas, así como as estructuras de control mais habituais utilizadas nos modelos; no apartado (§3.3) descríbense os aspectos relacionados coa semántica do Grafcet, as regras de evolución, as interpretacións e postulados temporais que garanten a reactividade e determinismo dos modelos e a interpretación temporal das accións; no apartado (§3.4) inclúense algúns exemplos de modelado con Grafcet; no apartado (§3.6) introdúcese o estándar IEC 61131-3 e o SFC; e finalmente, o capítulo resúmese en (§3.7).

3.2. A sintaxe do Grafcet

3.2.1. Elementos básicos

O Grafcet é un grafo orientado formado por dous tipos diferentes de nodos: as *etapas* e as *transicións*. Os nodos únense mediante *arcos orientados*. A única regra sintáctica definida para o Grafcet é a que impide que dúas etapas ou dúas transicións estean conectadas entre si.

3.2.1.1. Etapas

As etapas representan comportamentos invariantes dentro da secuencia lóxica que describe o funcionamento do sistema de control e represéntanse graficamente mediante un cadrado (Figura 3.1.a). Durante a interpretación do modelo cada etapa poderá estar nun de dous estados posíbeis: activa ou inactiva. O conxunto de etapas activas nun instante dado determina a *situación* do modelo. Para indicar que unha etapa está activa utilizase unha marca distintiva colocada no interior do cadrado, normalmente un pequeno círculo (Figura 3.1.b). Para diferenciar as *etapas iniciais,* aquelas que estarán activas ao comezo da interpretación do modelo, utilizase un cadrado con dobre liña (Figura 3.1.c). As etapas son identificadas mediante un valor alfanumérico situado no interior do cadrado. Cada etapa terá asociada unha variábel booleana, denominada X_i (sendo i o identificador alfanumérico da etapa) que durante a interpretación do modelo terá un valor igual a *true* sempre que a etapa estea activa, e igual a *false* cando estea inactiva. As etapas poden ter asociadas accións (§3.2.1.5), que só serán executadas cando a etapa estea activa.

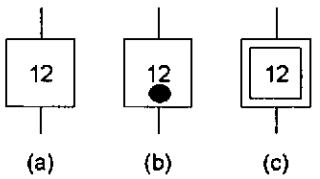


Figura 3.1. Representación gráfica de: (a) unha etapa inactiva; (b) unha etapa activa; (c) unha etapa inicial.

3.2.1.2. Transicións

As transicións representan posíbeis evolucións da secuencia de control entre etapas e son representadas graficamente mediante un trazo groso horizontal (Figura 3.2). Cada transición pode ter asociada unha función booleana denominada *receptividade* que determinará, dacordo ás regras de evolución do Grafcet (§3.3.1), cando unha transición poderá ser franqueada (é dicir, cando pode producirse unha evolución na situación do modelo). Nunca pode haber máis dunha transición entre dúas etapas calquera e a súa identificación faise mediante un valor alfanumérico que pode aparecer entre parénteses á esquerda da transición.

Figura 3.2. Representación gráfica dunha transición.

3.2.1.3. Arcos orientados

Os arcos orientados (Figura 3.3) deben unir sempre unha etapa a unha transición ou viceversa, mais nunca dúas etapas ou dúas transicións entre si. O sentido do arco pode indicarse mediante unha frecha nun dos seus extremos, sendo aconsellábel utilizalo sempre que o arco vaia de abaixo cara riba.

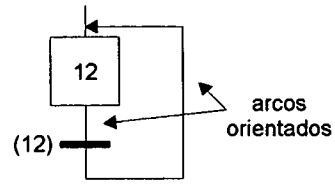


Figura 3.3. Representación gráfica dos arcos orientados.

3.2.1.4. Receptividades

As receptividades son condicións lóxicas booleanas asociadas ás transicións que aparecen representadas de maneira textual ou simbólica á dereita das transicións (Figura 3.4). O resultado da avaliación da condición determinará se unha transición validada pode ou non ser franqueada e, en consecuencia, se a situación do modelo pode evoluir (§3.3.1). Nas receptividades poden utilizarse variábeis internas, externas e eventos relacionados mediante os operadores booleanos AND, OR e NOT. As *variábeis internas* son as que representan os valores de activación das etapas e almacenan cálculos internos. As *variábeis externas* son as que representan os valores das variábeis do proceso controlado, os sinais de mando proporcionados polo operador, o estado dos temporizadores e contadores, etc. En xeral, e dende un punto de vista puramente formal, pode considerarse como variábel externa calquera sinal que teña un caracter asíncrono en relación á evolución do Grafcet, e como interna a modificada de forma síncrona coa evolución do modelo. Os *eventos*, representados mediante os operadores ↑ e ↓, indican a ocorrencia dun cambio no valor dunha variábel ou condición lóxica e clasifícanse en externos e internos segundo afecten a variábeis externas ou internas respectivamente.

Figura 3.4. Representación gráfica dunha transición con receptividade asociada.

3.2.1.5. Accións

As accións están asociadas ás etapas e describen a forma en que son modificadas as variábeis de saída do modelo cando as etapas están activas. Represéntanse graficamente (Figura 3.5) como unha descrición textual ou simbólica no interior dun rectángulo unido mediante unha liña á etapa. Cada acción está asociada a unha única etapa, mais unha mesma etapa pode ter asociadas calquera número de accións. Cando isto acontece únense os rectángulos das accións en horizontal ou vertical (Figura 3.6), sen que esta representación implique ningunha relación de orde ou prioridade entre elas. As etapas sen accións asociadas utilízanse normalmente para modelar estados de espera ou sincronización. Cada acción está composta por tres seccións [84]:

1. Unha sección opcional na que se indica o *tipo de acción*. Este tipo determina a relación temporal entre o estado de activación da etapa e os valores das saídas modificados pola acción (§3.3.3). O tipo indícase mediante unha ou varias das letras da Táboa 3-I.
2. Unha sección que contén a *descrición da acción*. O formato concreto desta sección non é especificado nas normas, de xeito que poida utilizarse calquera formalismo que se considere axeitado: cronogramas, organigramas, funcións de transferencia, ecuacións de estado, etc. A norma IEC 61131-3 é un exemplo de proposta dunha sintaxe para a especificación dos contidos desta sección.
3. Unha sección opcional que contén o *identificador* da variábel que será utilizada para indicar a finalización da acción.

Letra	Tipo de acción
N	Continua
R, S	Memorizada
D	Demorada
L	Limitada
P	Impulsional
C	Condicional

Táboa 3-I. Tipos de accións.

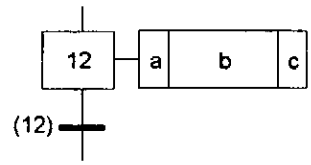


Figura 3.5. Representación gráfica dunha acción coas tres seccións definidas polo estándar: (a) tipo; (b) descrición; e (c) identificación.

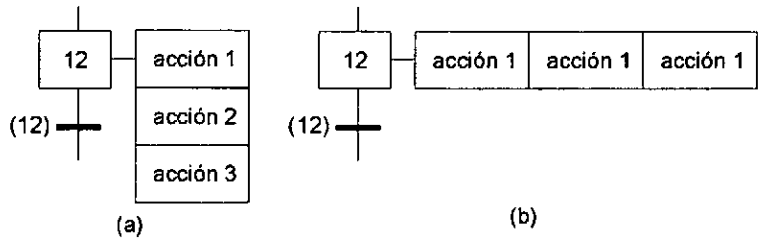


Figura 3.6. Representación gráfica de varias accións asociadas á mesma etapa: (a) en vertical; e (b) en horizontal.

3.2.2. Extensións sintácticas

Para facilitar a estructuración dos modelos de sistemas complexos propuxéronse varias extensións sintácticas á proposta inicial recollidas en [165]. Estas propostas permiten a aplicación de estratexias de deseño tipo 'divide e vencerás', consistente na división dun problema complexo en múltiples subproblemas mais sinxelos; ou "top-down", consistente en concentrarse nos aspectos relevantes a un nivel de abstracción deixando os detalles superfluos para a súa consideración posterior. As extensións sintácticas explicadas nesta sección son as seguintes: etapas fonte e sumidoiro (§3.2.2.1), macroetapas (§3.2.2.2), particións (§3.2.2.3) e ordes de forzado (§3.2.2.4).

3.2.2.1. Etapas fonte e sumidoiro

Unha *etapa fonte* (Figura 3.7.a) é unha etapa que unicamente ten transicións conectadas sucedéndoa na secuencia de control. As etapas fonte só poden activarse pola aplicación da regra 1 de evolución (§3.3.1.1), se son etapas iniciais, ou ben ser forzadas dende un grafcet parcial de nivel xerárquico superior (§3.2.2.4). Da mesma maneira, unha *etapa sumidoiro* (Figura 3.7.b) é unha etapa que unicamente ten transicións conectadas precedéndoa na secuencia de control. Estas etapas só son desactivadas en caso de que a situación do grafcet ao que pertencen sexa forzada.

De xeito semellante son definidas as *transicións fonte* e *sumidoiro* (Figura 3.8), cunha pequena diferencia: unha etapa pode ser ao mesmo tempo fonte e sumidoiro mais non unha transición, xa que isto non tería sentido dende un punto de vista lóxico. As transicións sen

conexións a etapas non afectan á evolución da secuencia de control. A única consecuencia da súa utilización sería a introducción dunha sobrecarga durante a interpretación do modelo, pois considérase que as transicións fonte están sempre validadas e, en consecuencia, a súa receptividade será avaliada en cada ciclo do algoritmo de interpretación utilizado (§3.3.2.2).

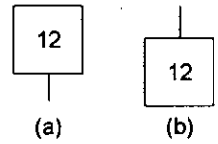


Figura 3.7. Representación gráfica de etapas: (a) fonte; e (b) sumidoiro.

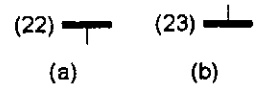


Figura 3.8. Representación gráfica de transicións: (a) fonte; e (b) sumidoiro.

3.2.2.2. Macroetapas

As *macroetapas* (Figura 3.9) son representacións simplificadas dun conxunto de etapas e transicións, denominado *expansión da macroetapa*, que cumpre as seguintes regras sintácticas:

1. Hai unha única etapa de entrada e unha única etapa de saída, que se corresponden cos puntos de conexión da macroetapa no modelo.
2. O conxunto de etapas e transicións forman un grafo conexo (§3.2.2.3).
3. A cada macroetapa correspóndelle unha única expansión.

A utilización de macroetapas permite introducir diferentes niveis de detalle nos modelos e facilita a reutilización. As macroetapas ocultan os detalles superfluos nun nivel de descrición dado do modelo, que son especificados nun nivel inferior de detalle no interior das expansións. As macroetapas poden aniñarse (é posíbel utilizar novas macroetapas nas expansións doutras macroetapas) polo que é posíbel utilizar no modelo tantos niveis de detalle como se precisen.

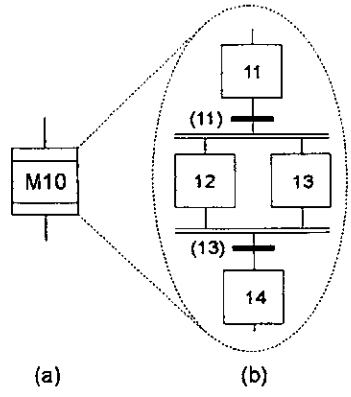


Figura 3.9. Representación gráfica dunha: (a) macroetapa; e (b) a súa expansión.

Durante a interpretación do modelo unha macroetapa estará activa cando o estea algunha das etapas da súa expansión. A utilización de macroetapas introduce distintos niveis de detalle na descrición do modelo, mais non na xerarquía de forzado (§3.2.2.4), do que se deduce que:

1. Dende un grafcet que conteña macroetapas non poden forzarse as etapas das súas expansións.

2. O contrario tampouco é posíbel, é dicir, dende a expansión dunha macroetapa non pode forzarse a situación do grafcet que contén a macroetapa.
3. Da mesma maneira tampouco poderá forzarse dende a expansión dunha macroetapa a situación dun grafcet de nivel xerárquico superior ao que contén a macroetapa.

3.2.2.3. Particións

En [165] defínense as seguintes particións dun modelo Grafcet:

1. Un *grafcet conexo* é aquel no que todo nodo está conectado directa ou indirectamente aos demais. Os grafcets conexos modelan secuencias simples que poden combinarse para formar estructuras complexas de nivel superior.
2. Un *grafcet parcial* é un conxunto de grafcets conexos. Os grafcets parciais modelan os subsistemas que forman un sistema complexo.
3. Un *grafcet global* é un conxunto de grafcets parciais. O grafcet global correspóndese co modelo do sistema de interese.

As particións definidas permiten estructurar o modelo do sistema formando unha *xerarquía estructural* (Figura 3.10), na que un grafcet global está formado por un ou varios grafcets parciais e estes, á súa vez, por un ou varios grafcets conexos que poden conter tantos niveis de detalle como se desexe mediante o uso de macroetapas (§3.2.2.2).

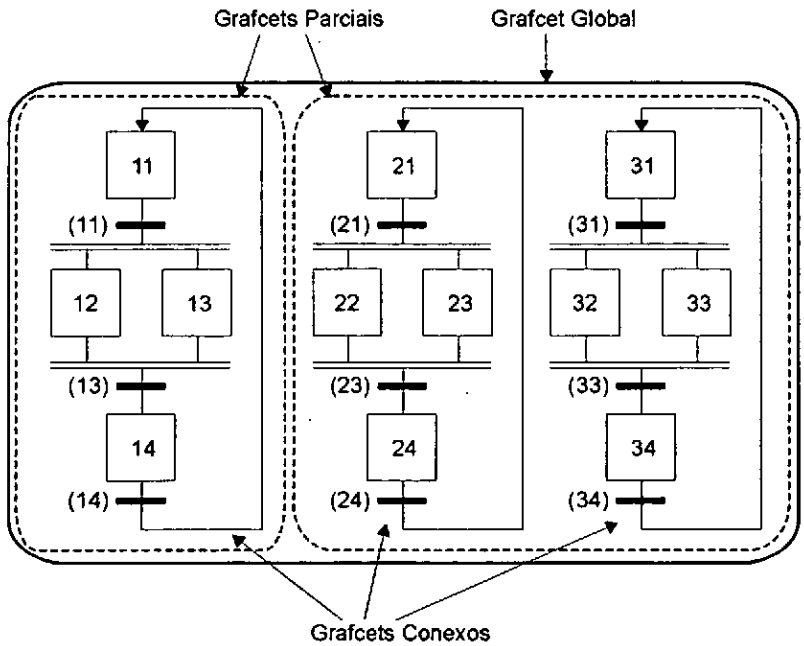


Figura 3.10. Xerarquía estructural do Grafcet.

3.2.2.4. Ordes de forzado

A definición dunha xerarquía estructural (§3.2.2.3) non é suficiente para modelar as relacións existentes entre as distintas compoñentes dun sistema. É preciso dispor ademais de mecanismos que permitan especificar relacións máis complexas. As *ordes de forzado* [103] foron incluídas como parte da definición do Grafcet para modelar as relacións que se establecen nun sistema cando o estado dalgún dos seus subsistemas é controlado por outros. Desde o punto

de vista da estructuración lóxica do sistema, os subsistemas dependentes son considerados como pertencentes a un nivel inferior ao dos subsistemas dos que dependen.

As ordes de forzado considéranse como ordes internas que, ao igual que as accións, están asociadas ás etapas. Estas ordes son utilizadas para forzar dende un grafcet parcial a situación doutro, que se manterá na situación forzada mentres a etapa á que a orde de forzado estea asociada siga activa. Exemplos de utilización das ordes de forzado son a activación de secuencias para a detección de faios e a xestión de situacións de emerxencia, a especificación dos estados de operación dun sistema, etc.

A representación gráfica das ordes de forzado é semellante á das accións (Figura 3.5) excepto pola sintaxe utilizada para especificalas, que é a seguinte:

$$F/G_n : \{situación\}$$

sendo:

- F/ o indicador utilizado para identificar unha orde de forzado.
- G_n o identificador do grafcet parcial forzado.
- *{situación}* a situación que se quere forzar.

A utilización de ordes de forzado no modelado dun sistema establece relacións de dependencia entre os grafcets parciais forzados e os 'forzadores' que en conxunto forman unha *xerarquía de forzado* (Figura 3.11). Os grafcets parciais forzados están situados nesta xerarquía en niveis inferiores aos dos seus 'forzadores'.

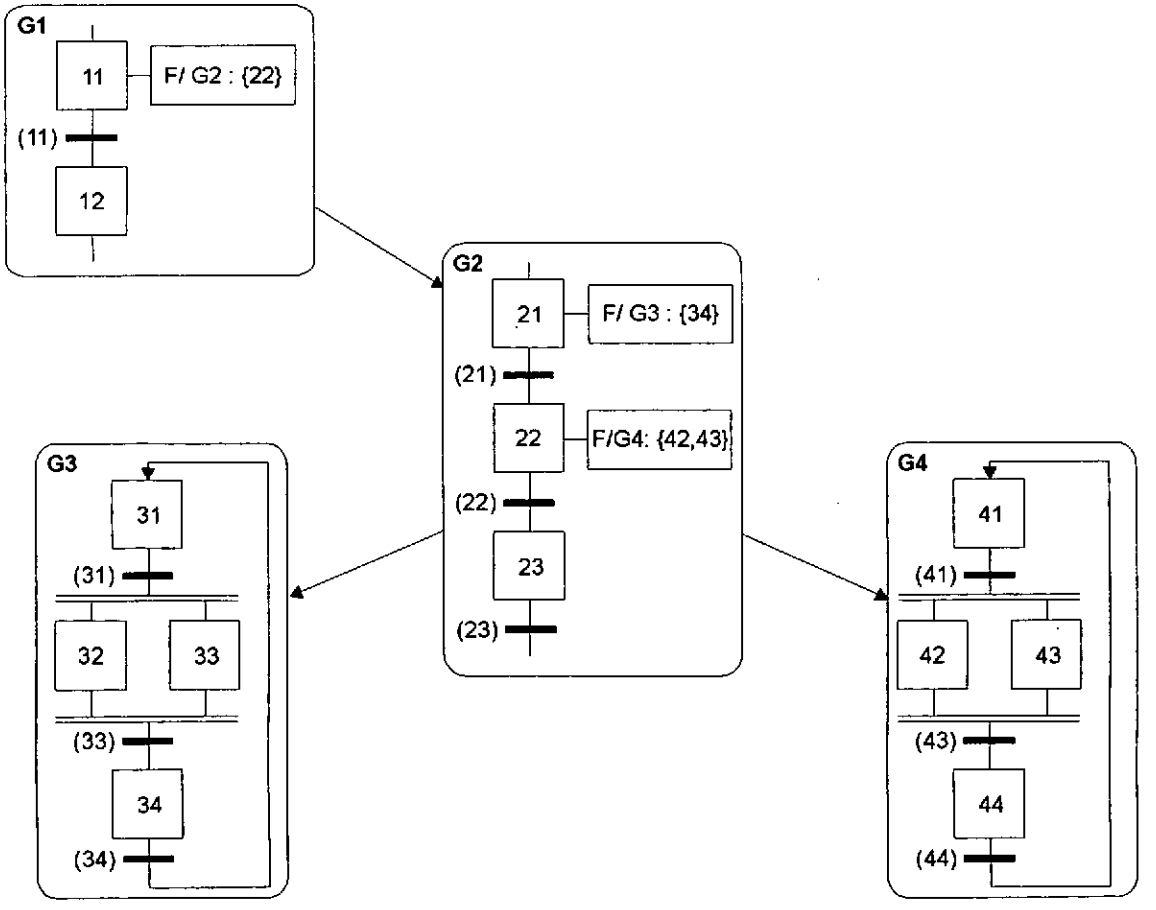


Figura 3.11. Exemplo dunha xerarquía de forzado.

Para manter a coherencia dun modelo estructurado desta maneira e garantir o determinismo da súa interpretación, a xerarquía de forzado ten que cumprir as regras seguintes:

1. A xerarquía ten que ser total, é dicir, se un grafcet parcial forza directa ou indirectamente a outro, o contrario non pode acontecer. Esta regra garante que a xerarquía de forzado non conteña ciclos e que os grafcets parciais 'forzadores' estean situados en niveis superiores aos forzados. En [103] proponse un método baseado na teoría de grafos para comprobar a coherencia dunha xerarquía de forzado (§6.5.1.3).
2. Durante a interpretación do modelo e para todo instante de funcionamento cada grafcet parcial forzado terá un único 'forzador'. Esta regra evita os conflictos entre ordes de forzado durante a interpretación do modelo.

Estas regras complétanse coa modificación do algoritmo de interpretación para ter en conta a aplicación das ordes de forzado (§3.3.2.5) respectando a xerarquía de forzado e garantindo o determinismo e a coherencia da interpretación.

3.2.3. Estructuras de control básicas

Nesta sección recóllense algunhas das estructuras de control máis comúns nos modelos Grafcet e como son representadas graficamente.

3.2.3.1. Secuencia

Unha *secuencia* está formada por un conxunto de etapas e transicións conectadas de maneira que a continuación de cada etapa só pode haber unha única transición e a continuación de cada transición unha única etapa (Figura 3.12).

3.2.3.2. Selección de secuencia

A *selección de secuencia,* indicada graficamente mediante unha liña horizontal (Figura 3.13), é unha estructura que permite especificar unha alternativa entre varias posíbeis secuencias, que estarán conectadas a continuación dunha ou máis etapas por medio de tantas transicións como alternativas haxa. Para evitar situacións non desexadas de paralelismo interpretado (§3.2.3.10) —activación de máis dunha secuencia despois dunha selección— durante a interpretación do modelo debe garantirse que as condicións asociadas ás transicións da selección son exclusivas entre si.

3.2.3.3. Fin de selección de secuencia

O *fin de selección de secuencia* (Figura 3.14) representa o punto de unión sen sincronización de varias secuencias de control. Cada secuencia finaliza cunha transición e, se se garantiu a exclusividade no inicio da selección, só unha das secuencias que converxen estará activa durante a interpretación do modelo.

3.2.3.4. Paralelismo

O *paralelismo*, indicado graficamente mediante unha dobre liña horizontal (Figura 3.15), é unha estructura que representa a activación simultánea de varias secuencias concorrentes introducindo nos modelos o concepto de paralelismo estructural (§3.2.3.10). Neste caso haberá unha única transición entre as secuencias a activar e a etapa ou etapas que as anteceden.

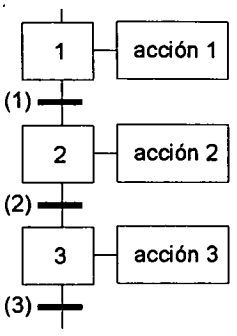


Figura 3.12. Representación gráfica dunha secuencia.

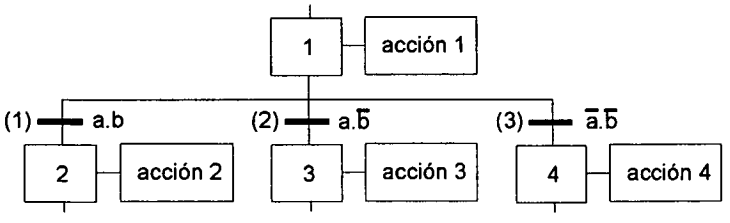


Figura 3.13. Representación gráfica dunha selección de secuencia.

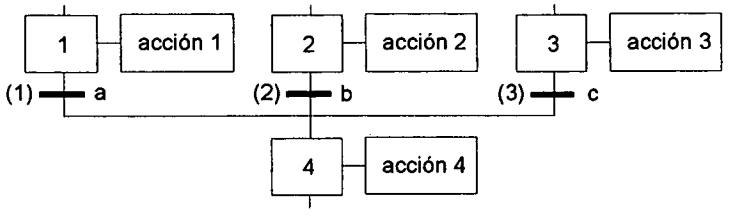


Figura 3.14. Representación gráfica do final dunha selección de secuencia.

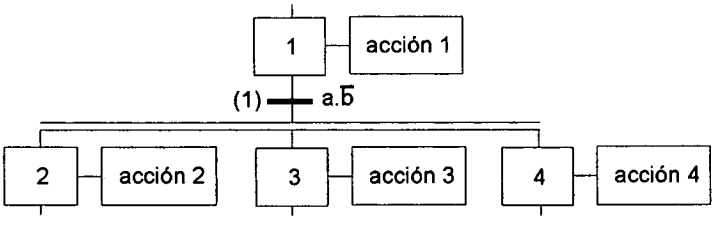


Figura 3.15. Representación gráfica do comezo de varias secuencias paralelas.

3.2.3.5. Fin de paralelismo (sincronización)

O *fin de paralelismo* é unha estructura complementaria coa anterior que permite representar un punto de sincronización entre varias secuencias concorrentes, que estarán conectadas mediante unha única transición á etapa ou etapas que as suceden (Figura 3.16). Durante a interpretación do modelo, e dacordo á segunda regra de evolución (§3.3.1.2), esta transición só estará validada cando todas as etapas que a preceden estean activas (Figura 3.17).

3.2.3.6. Salto de etapas

O *salto de etapas* permite avanzar a situación dunha secuencia 'saltando' algunha das etapas que a forman. Esta estructura pode considerarse como un caso particular da selección de secuencia (§3.2.3.2) na que só hai dúas posíbeis escollas excluíntes entre si: ou ben continuar a secuencia ou ben avanzar ata unha etapa posterior. O salto de etapas representase graficamente mediante unha transición que une as etapas de inicio e destino do 'salto' (Figura 3.18.a).

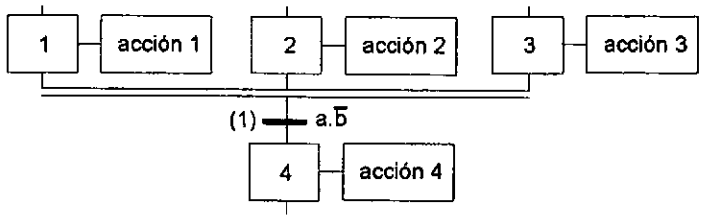


Figura 3.16. Representación gráfica do final de varias secuencias paralelas.

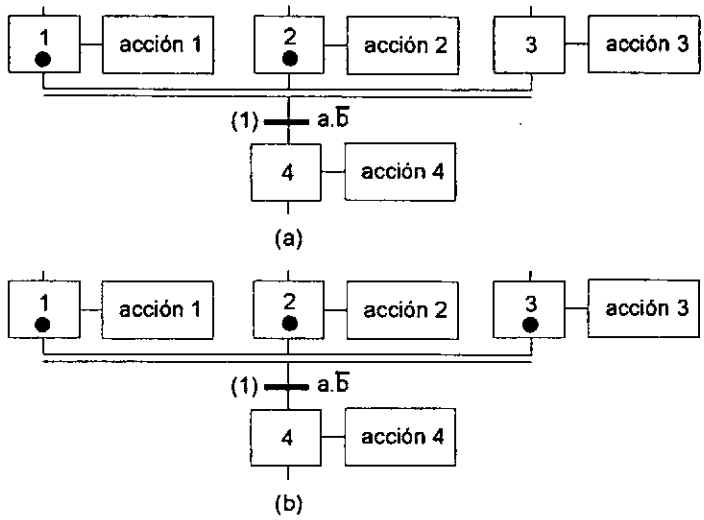


Figura 3.17. Exemplo de sincronización: (a) a transición 1 non está validada; (b) a transición 1 si está validada.

3.2.3.7. Ciclo

Un *ciclo* é o equivalente a un salto a un punto anterior na secuencia. Represéntase graficamente coma un salto, indicando o sentido cunha frecha no arco orientado (Figura 3.18.b).

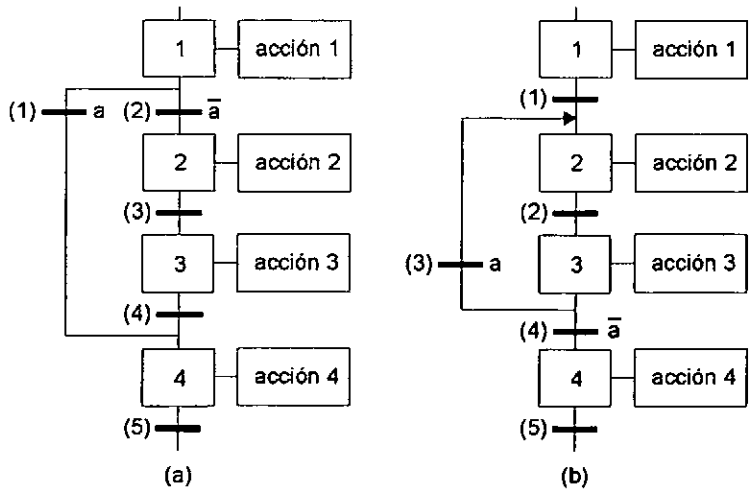


Figura 3.18. Representación gráfica de: a) salto de etapas; e b) un ciclo.

3.2.3.8. Semáforo

O *semáforo* permite representar a exclusión mutua no acceso a un recurso compartido desde varias secuencias. Exemplos destes recursos son os postos de almacenaxe de pezas, os carros de

transporte automatizados, etc. O semáforo garante o acceso exclusivo ao recurso compartido a unha soa das secuencias. Nos modelos os semáforos son representados mediante etapas iniciais situadas en paralelo coa finalización das secuencias que acceden ao recurso compartido e sincronizadas co seu comezo (Figura 3.19).

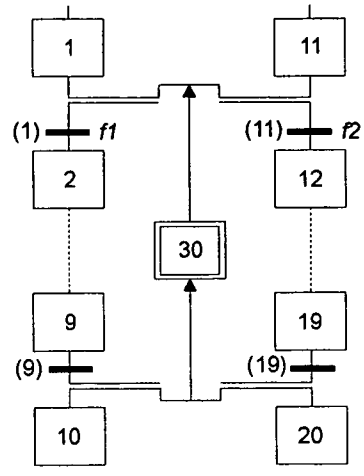


Figura 3.19. Representación gráfica dun semáforo —etapa 30— que proporciona un mecanismo de exclusión mutua entre dúas secuencias.

O funcionamento do semáforo durante a interpretación do modelo é o seguinte: inicialmente a etapa que modela o semáforo —etapa 30— estará activa, cando as secuencias que acceden ao recurso vaian comezar simultaneamente —etapas 1 e 11 activas—, as transicións que lles dan comezo —transicións 1 e 11— estarán validadas, entón a primeira das receptividades asociadas a estas transicións —condicións f_1 e f_2— que sexa certa determinará qué secuencia accede ao recurso compartido. Cando isto acontece (supoñamos que f_1 é a receptividade franqueada) a etapa inicial da secuencia —etapa 2— actívase e desactívanse as etapas 1 e 30 (o semáforo), co que a transición 11 deixa de estar validada impedindo que outra secuencia poida acceder simultaneamente ao recurso compartido. Unha vez remate o acceso ao recurso actívanse as etapas 10 e 30 e a transición 11 volve estar validada, e a segunda secuencia xa pode acceder ao recurso cando a receptividade f_2 sexa certa. É importante destacar que para que o semáforo funcione correctamente é preciso que as receptividades f_1 e f_2 sexan exclusivas. A estructura dun semáforo pode ampliarse a calquera número de secuencias, como mostra a Figura 3.20.

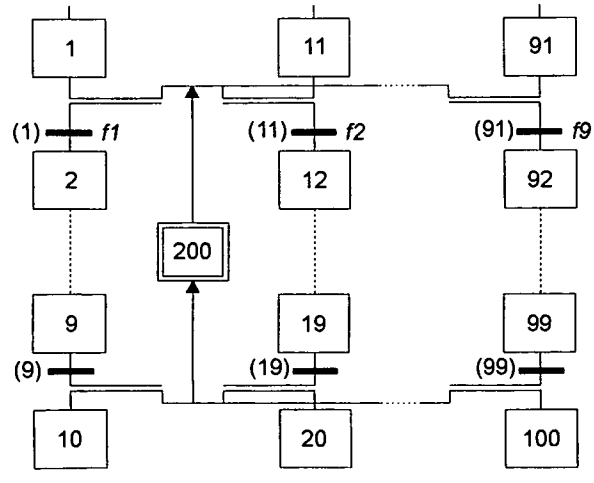


Figura 3.20. Representación gráfica dun semáforo —etapa 200— que proporciona un mecanismo de exclusión mutua entre varias secuencias.

Unha característica do semáforo é que a orde de activación das secuencias ou o número de veces que se activa cada unha non é significativo. Cando é preciso establecer unha orde de activación pode utilizarse unha variante como a da Figura 3.21. Neste caso as secuencias acceden a un recurso compartido executando operacións que deben realizarse de maneira alterna. Na Figura 3.22 móstrase un exemplo no que se modela a sincronización das operacións realizadas por dúas máquinas, unha de producción de pezas e outra de ensamblaxe, comunicadas mediante un posto de almacenaxe no que só pode haber unha peza. O depósito e a retirada dunha peza no posto de almacenaxe son operacións exclusivas que se realizan alternativamente comezando pola de depósito da peza.

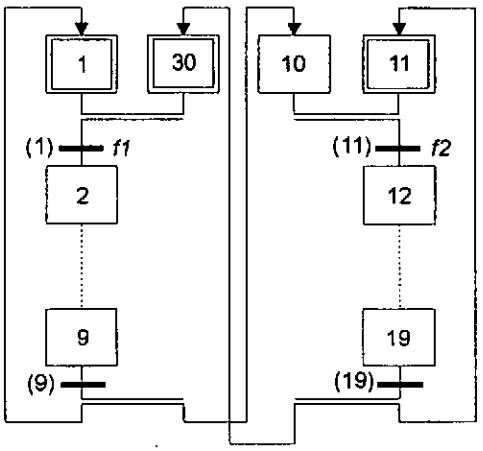


Figura 3.21. Representación gráfica dunha estructura que proporciona un mecanismo de alternancia entre dúas secuencias de accións.

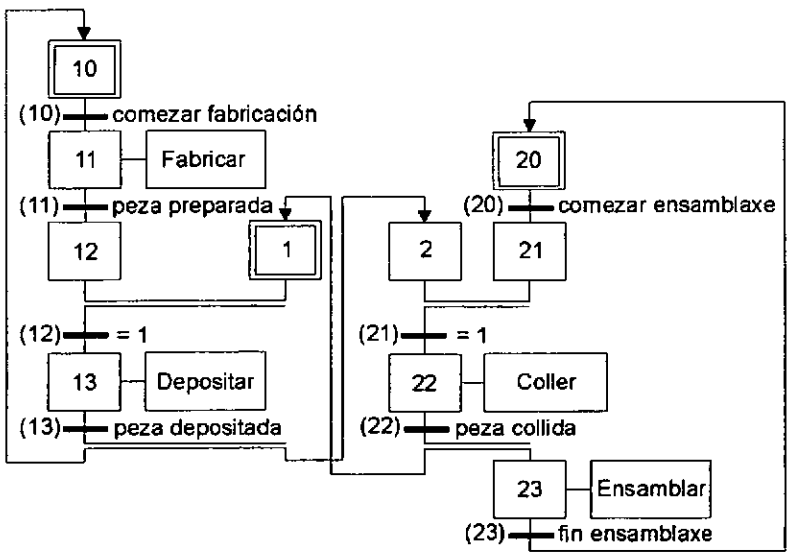


Figura 3.22. Exemplo de alternancia das secuencias de fabricación e ensamblaxe, coordinadas mediante as operacións de depósito e recollida de pezas.

3.2.3.9. Acumulación e reserva

A *acumulación* (Figura 3.23.a) utilizase para representar a sincronización de múltiples activacións, coincidentes ou non no tempo, dunha mesma secuencia. A *reserva* ou *xeración* (Figura 3.23.b) utilizase para representar múltiples activacións simultáneas dunha mesma secuencia.

O funcionamento da acumulación durante a interpretación do modelo é o seguinte: a transición 3 só estará validada cando as etapas 1, 2, e 3 estean activas, a primeira vez que se active a secuencia as receptividades das transicións 1 e 2 son certas, polo que se activará a etapa 3. A segunda vez a transición 2 será falsa e a secuencia rematará activando a etapa 2, e a terceira vez a transición 1 será falsa e a secuencia rematará activando a etapa 1. Nese intre a transición 3 estará validada e cando a súa receptividade sexa certa desactivaranse as etapas 1, 2 e 3 e continuarase coa interpretación do modelo.

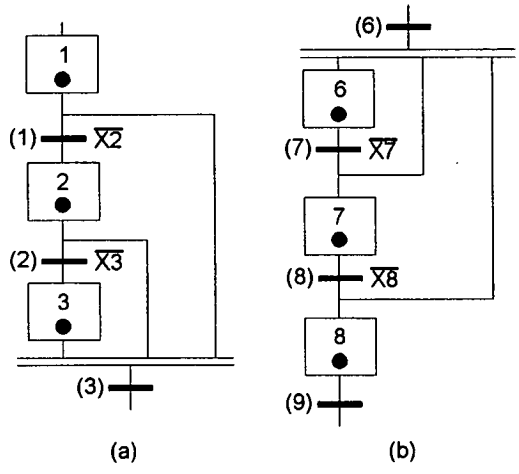


Figura 3.23. Representación gráfica dunha estructura que realiza: a) unha acumulación; e b) unha reserva.

No referente á reserva, cando a transición 6 é franqueada actívanse simultaneamente as etapas 6, 7 e 8, e valídanse as transicións 7, 8 e 9. Nesta situación as receptividades das transicións 7 e 8 serán falsas, e cando a receptividade da transición 9 pase a ser certa desactivarase a etapa 8, a receptividade da transición 8 pasará a ser certa co que se activará de novo a etapa 8 e desactivarase a 7. Isto á súa vez provoca que a receptividade da transición 7 pase a ser certa e sexa reactivada a etapa 7 e desactivada a 6. Na nova situación estarán activas as etapas 7 e a 8, validadas as transicións 8 e 9 e será falsa a receptividade da transición 8. Cando a receptividade da transición 9 volva ser certa repetirase o proceso anterior. Esta estructura garante que a secuencia posterior á transición 9 será activada tres veces.

3.2.3.10. Paralelismo interpretado e paralelismo estructural

Como se indicou en (§3.2.3.4) a estructura de paralelismo permite representar explicitamente nos modelos Grafcet a activación simultánea de dúas ou mais secuencias concorrentes. Esta representación explícita é denominada *paralelismo estructural*, en contraposición ao *paralelismo interpretado*, que consiste na representación nos modelos de estructuras que activan simultaneamente varias secuencias concorrentes sen indicalo co símbolo gráfico utilizado para representar o paralelismo estructural (Figura 3.15). Se isto se fai inadvertidamente pode dar lugar durante a interpretación do modelo a comportamentos imprevistos polo que, na medida do posíbel, debe evitarse esta situación.

As estructuras que inician unha situación de paralelismo interpretado denomínanse *conflictos*, e caracterízanse por estar formadas por dúas ou mais transicións con receptividades que non son mutuamente exclusivas e cuxa activación depende de etapas comúns. O exemplo mais simple de conflicto é unha selección de secuencia (Figura 3.13) na que as receptividades das transicións implicadas non son mutuamente exclusivas. A eliminación de conflictos debe ter en conta a semántica que pretenda modelarse, de xeito que se o que se pretende é non

permitir a activación simultánea de varias secuencias entón deberán modificarse as receptividades para facelas mutuamente exclusivas. Polo contrario, se o que se pretende é permitir a activación simultánea de varias secuencias entón deberá modificarse a estructura para representar explicitamente o paralelismo estructural.

3.3. A semántica do Grafcet

Na proposta inicial do Grafcet [1] ademais de definirse os elementos sintácticos e as regras que permiten combinalos para construír modelos sintacticamente correctos, definíronse tamén as regras para a interpretación semántica dos modelos denominadas *regras internas de evolución*, que permiten calcular as evolucións dun modelo dado, determinándose a situación seguinte do modelo a partires da súa situación actual e dos valores das variábeis de entrada. Nesta sección preséntanse estas regras, explícanse as diferentes interpretacións semánticas ás que poden dar lugar e descríbese o comportamento temporal dos distintos tipos de accións utilizados nos modelos.

3.3.1. As regras de evolución

A *situación inicial* dun modelo establécese aplicando a seguinte regra:

3.3.1.1. Regra 1: Situación inicial

Inicialmente activaranse unicamente as etapas iniciais do grafcet, permanecendo as demais etapas inactivas ata que se produza unha evolución na situación do modelo (Figura 3.24).

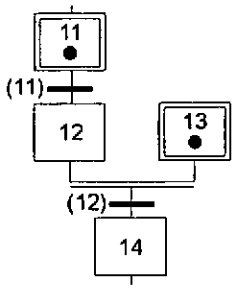


Figura 3.24. Exemplo de activación inicial dun Grafcet.

A evolución do modelo dunha situación á seguinte realizase mediante o franqueamento de transicións dacordo ás regras seguintes:

3.3.1.2. Regra 2: Determinación das transicións franqueábeis

Unha transición será *franqueábel* cando se cumpran as dúas condicións seguintes (Figura 3.25):

1. Todas as etapas que anteceden á transición están activas. Neste caso dirase que a transición está *validada*.
2. A receptividade asociada á transición é certa.

De cumprirse ambas as dúas condicións a transición deberá ser franqueada, o que producirá unha evolución na situación do modelo.

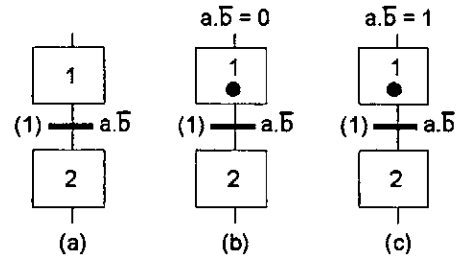


Figura 3.25. Estados dunha transición: (a) transición non validada; (b) transición validada, e (c) transición validada e franqueábel.

3.3.1.3. Regra 3: Franqueamento dunha transición

O franqueamento dunha transición consiste na desactivación das etapas que a anteceden e a activación das etapas que a suceden (Figura 3.26). A desactivación e activación destas etapas é simultánea.

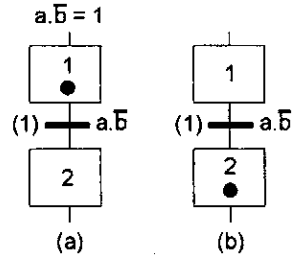


Figura 3.26. Franqueamento da transición 1: (a) situación anterior; e (b) situación posterior.

Debido a que o Grafcet aplícase ao modelado de sistemas físicos complexos, as evolucións do modelo implican normalmente o franqueamento simultáneo de múltiples transicións. Para garantir o *sincronismo* das evolucións definiuse a regra seguinte:

3.3.1.4. Regra 4: Evolucións simultáneas

Se durante unha evolución do modelo hai varias transicións franqueábeis, todas elas serán franqueadas simultaneamente (Figura 3.27).

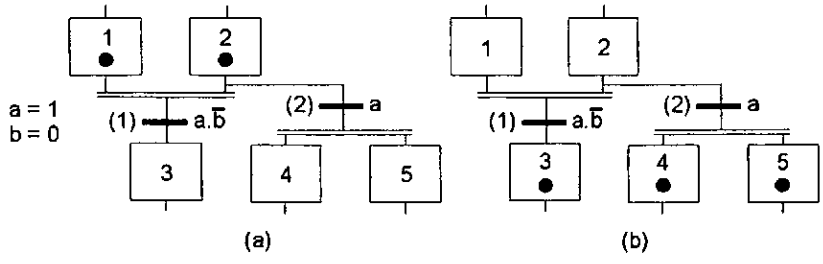


Figura 3.27. Franqueamento simultáneo das transicións 1 e 2: (a) situación anterior; e (b) situación posterior.

Por último definiuse unha regra que se aplica cando unha etapa é activada e desactivada simultaneamente durante unha evolución do modelo:

3.3.1.5. Regra 5: Activación e desactivación simultánea dunha etapa

Se durante unha evolución do modelo, unha etapa é activada e desactivada simultaneamente, permanecerá activa na nova situación.

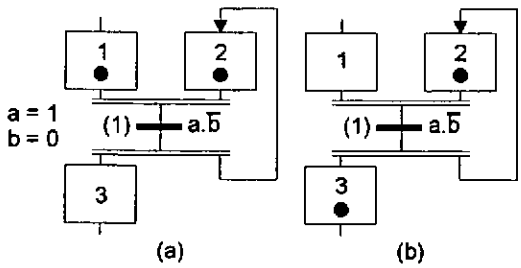


Figura 3.28. Activación e desactivación simultánea da etapa 2: (a) situación anterior; e (b) situación posterior.

3.3.2. Semántica temporal do modelo

A práctica posterior á definición das regras de evolución do Grafcet demostrou que non abondaba co definido para permitir a implementación práctica dos modelos sen ambigüidades na interpretación. Para resolver este problema presentouse unha segunda proposta [2] que complementaba á primeira definindo por unha banda os postulados temporais que aclaran a relación entre as evolucións internas do modelo e o contexto temporal dictado polo seu contorno e, por outra, as diferentes interpretacións dos modelos que poden derivarse da aplicación das regras de evolución.

3.3.2.1. Postulados temporais

Os postulados que especifican a relación temporal entre as evolucións internas dun modelo —dacordo ás regras de evolución explicadas en (§3.3.1)— e o seu contorno son os seguintes:

Postulado 1

A duración do franqueamento dunha transición (e, en consecuencia, o de activación dunha etapa) pode ser, dende un punto de vista teórico, tan pequeno como se desexe, mais non pode ser nulo. Na práctica este tempo dependerá da tecnoloxía empregada na implementación física do sistema.

Este postulado ten consecuencias directas na interpretación das regras de evolución relacionadas co franqueamento simultáneo de transicións. Debido a que o postulado establece que a duración do franqueamento dunha transición non pode ser nulo, cando unha etapa se activa as transicións que a suceden non serán franqueadas inmediatamente —en aplicación da regra 4 (§3.3.1.4)— aínda que as súas receptividades sexan certas. O motivo é que ao non ser nulo o tempo de franqueamento das transicións previas á etapa, a activación da etapa non é instantánea e, polo tanto, as transicións que a suceden non estarían validadas. O franqueamento desas transicións faríase nunha evolución posterior do modelo se as súas receptividades seguiran sendo certas. Outra consecuencia directa deste postulado é que as accións asociadas ás etapas activas son consideradas durante un tempo teoricamente tan pequeno como se desexe mais non nulo.

Postulado 2

Os modelos Grafcet baséanse na hipótese da existencia dun contorno asíncrono coa restricción de que calquera dous eventos externos non relacionados non poden acontecer simultaneamente.

Unha consecuencia directa deste segundo postulado é o establecemento dun límite superior para a duración dunha evolución interna do modelo. Este límite ven dado pola separación mínima entre dous eventos externos calquera non relacionados. Teoricamente, se todas as

evolucións internas do modelo poden facerse nun tempo inferior a este límite garántese a *reactividade* (capacidade de resposta ante todos os eventos externos que se produzan) do modelo.

3.3.2.2. Algoritmos de interpretación do Grafcet

Ademais dos postulados temporais, na revisión da definición do Grafcet propuxéronse dous posíbeis algoritmos de interpretación distintos que utilizan as regras de evolución do modelo (§3.3.1): *sen busca da estabilidade* (SRS) e *con busca da estabilidade* (ARS). Estes algoritmos proporcionan unha referencia teórica que pode servir de guía na implementación dun intérprete Grafcet considerando as restricción dunha tecnoloxía específica. A utilización dun ou doutro algoritmo dependerá dos requirimentos de *reactividade* e *determinismo* (a toda secuencia de variación das entradas lle corresponde unha única secuencia de variacións das saídas) do sistema, e das restricción da tecnoloxía utilizada.

Así para unha aplicación específica, se fose posíbel garantir que o tempo preciso para pasar dunha *situación estábel* (situación dende a que só é posíbel evoluir debido a ocorrencia dun evento externo) á seguinte sempre é inferior ao intervalo mínimo entre dous eventos externos non relacionados, entón o algoritmo SRS garante os requisitos de determinismo e reactividade[23]. Sen embargo na práctica non é posíbel garantir esa condición, e o algoritmo SRS pode resultar nunha interpretación non determinista do modelo dependendo da relación existente entre o tempo preciso para realizar unha evolución interna do modelo e o intervalo mínimo entre dous eventos externos non relacionados. Como pode verse no exemplo da Figura 3.29, se se considera a situación inicial {1} como situación de partida, o franqueamento da transición 1 producirase coa ocorrencia do evento $\uparrow a$ e a evolución do modelo dependerá de se o evento $\downarrow a$ ocorre antes ou despois da activación da etapa 2. A demostración é a seguinte: sexa $t_a = t(\downarrow a) - t(\uparrow a)$ o tempo transcorrido entre ambos eventos e t_f o tempo consumido no franqueamento da transición 1; entón se $t_f < t_a$ a situación seguinte será {3}, pois o valor da variábel *a* será 1 cando se avalíen as condicións asociadas ás transicións 2 e 3, polo contrario, se $t_f > t_a$ a situación final será {4}, pois nese caso o valor da variábel *a* será 0.

O segundo dos algoritmos propostos, o algoritmo ARS, resolve en parte este problema. Este algoritmo garante o determinismo da interpretación do modelo independentemente do tempo de franqueamento das transicións, mais co custe de degradar a reactividade. A solución aportada por este algoritmo consiste en ter en conta os eventos externos só cando o modelo está nunha situación estábel, o que implica que os eventos que se produzan durante a evolución interna do modelo (mentres pasa dunha situación estábel á seguinte) perderanse a menos que se proporcione algún mecanismo externo ao modelo para o seu almacenamento. O algoritmo ARS presenta un segundo problema, o *grafo de situacións* do modelo (grafo dirixido no que os estados representan as situacións do modelo e os arcos as evolucións) pode incluír *ciclos estacionarios* (evolucións compostas unicamente por situacións inestábeis). Nese caso a interpretación do modelo nunca alcanzaría unha situación estábel e o sistema quedaría bloqueado nunha busca sen fin dentro do ciclo.

3.3.2.3. Semiformalización da semántica do Grafcet: o xogador Grafcet

En traballos posteriores [25][106] á definición dos postulados temporais e algoritmos de interpretación, propúxose unha semiformalización dun intérprete de modelos Grafcet xenérico,

---

[23] Nótese que esta condición, xunto co postulado 2 comentado anteriormente, correspóndense co modelo de funcionamento no *modo fundamental* dos sistemas secuenciais asíncronos.

denominado *xogador Grafcet*, independente da implementación práctica utilizada. Este xogador constitúe unha guía de lectura ou interpretación da dinámica dos modelos que indica explicitamente e sen ambigüidades como deben integrarse conxuntamente as regras de evolución (§3.3.1), a lectura das entradas e escritura das saídas, os postulados temporais da relación do modelo co seu contorno (§3.3.2.1) e os algoritmos de interpretación (§3.3.2.2).

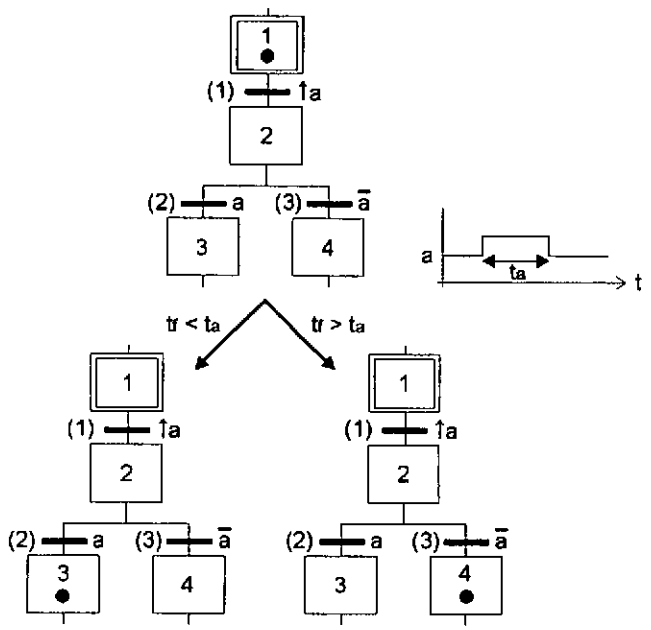


Figura 3.29. Demostración do non determinismo do algoritmo SRS.

O xogador mais elemental (Figura 3.30) basease nunha interpretación SRS dos modelos. Este xogador ten unha fase de iniciación na que se aplica a regra 1 (§3.3.1.1) para establecer a situación inicial do modelo, lense os valores iniciais das variábeis de entrada e calcúlanse os valores iniciais das variábeis de saída. Despois da fase inicial, o xogador entra nun ciclo no que se repiten as seguintes actividades:

1. Lectura dos valores das variábeis de entrada.
2. Cálculo das transicións franqueábeis —dacordo ao definido pola regra 2 (§3.3.1.2)—.
3. Evolución da situación franqueando as transicións franqueábeis —dacordo ao definido polas regras 3 e 4 (§3.3.1.3;§3.3.1.4)—, aplicando cando sexa preciso a regra 5 (§3.3.1.5) para a resolución dos conflictos de activación e desactivación simultánea de etapas.
4. Cálculo dos valores das variábeis de saída segundo o especificado polas accións asociadas ás etapas activas na nova situación do modelo.

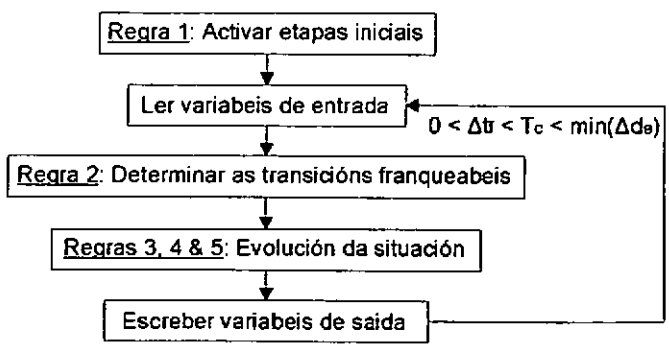


Figura 3.30. Xogador SRS de modelos Grafcet.

A consideración dos postulados temporais (§3.3.2.1) define ademais os límites temporais teóricos que delimitan o tempo de ciclo do xogador (T_c) para garantir o determinismo e a reactividade da interpretación en calquera implementación. O límite inferior será igual ao tempo preciso para o franqueamento das transicións: $\Delta t_f > 0$ (postulado 1), e o superior ao tempo mínimo entre dous eventos externos non relacionados: $\min(\Delta d_e) > 0$ (postulado 2). Da consideración conxunta dambos postulados obtense a seguinte desigualdade:

$$0 < \Delta t_f < T_c < \min(\Delta d_e) \tag{3.1}$$

A adaptación do xogador a unha interpretación ARS require a modificación do ciclo de actividades para representar o feito de que a lectura das entradas e a escritura das saídas faise só cando o modelo está nunha situación estábel. Deberá tamén engadirse unha nova actividade encargada da detección dos ciclos estacionarios que poida haber no grafo de situacións do modelo, para evitar que o xogador quede bloqueado nunha evolución sen fin. O ciclo de actividades modificado será o seguinte:

1. Lectura dos valores das variábeis de entrada.
2. Cálculo das transicións franqueábeis —dacordo ao definido pola regra 2 (§3.3.1.2)—.
3. Se hai algunha transición franqueábel, ir ao paso 6.
4. Cálculo dos valores das variábeis de saída segundo o especificado polas accións asociadas ás etapas activas na nova situación do modelo.
5. Volver ao paso 1.
6. Evolución da situación franqueando as transicións franqueábeis —dacordo ao definido polas regras 3 e 4 (§3.3.1.3;§3.3.1.4)—, aplicando cando sexa preciso a regra 5 (§3.3.1.5) para a resolución dos conflictos de activación e desactivación simultánea de etapas.
7. Se a nova situación non coincide con ningunha das xa visitadas neste ciclo de busca dunha situación estábel, ir ao paso 2.
8. Indicar á detección dun ciclo estacionario no grafo de situacións do modelo.

Como pode comprobarse, no xogador ARS existen realmente dous ciclos entrelazados. No primeiro deles, formado pola secuencia 2, 6, 7 e 8, realízase a evolución do modelo dende unha situación estábel á seguinte e detéctanse os ciclos estacionarios. Dentro deste ciclo non se len novos valores de entrada nin se obteñen novos valores de saída. A detección de ciclos estacionarios realízase comparando a situación actual do modelo coas situacións polas que xa se pasou durante o actual proceso de busca dunha situación estábel, se coincide con algunha entón a busca entrou nun ciclo estacionario. No segundo dos ciclos, formado pola secuencia 1, 2, 3, 4 e 5, comézase nunha situación estábel do modelo, lense os valores das variábeis de entrada, faise evoluir o modelo ata atopar unha nova situación estábel e calcúlanse os valores das saídas.

A consideración dos postulados temporais no xogador ARS modifica lixeiramente os límites do seu tempo de ciclo. Debido a existencia de dous ciclos, o tempo de ciclo total T_c é calculado agora como

$$T_c = T_{ci} + T_{ce} \tag{3.2}$$

sendo:

- T_{ci}, o tempo máximo que o ciclo interno precisa para calcular unha nova situación estábel.
- T_{ce}, o tempo máximo que o ciclo externo precisa para ler as entradas, calcular a nova situación do modelo e calcular as saídas.

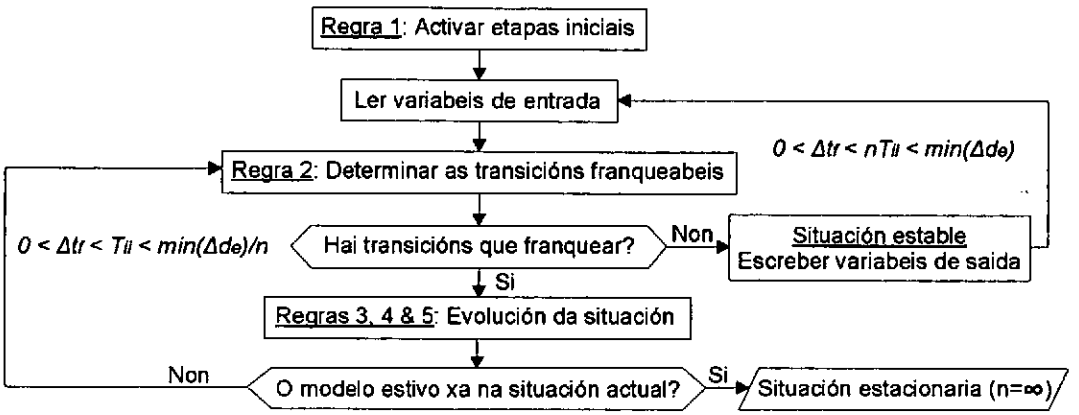


Figura 3.31. Xogador ARS de modelos Grafcet.

Pódese entón reescribir a inecuación (3.1) que limita o tempo de resposta do xogador como

$$0 < \Delta t_f < T_{ci} + T_{ce} < \min(\Delta d_e) \tag{3.3}$$

e debido a que

$$T_{ci} >> T_{ce} \tag{3.4}$$

e

$$T_{ci} = n\, T_{ii} \tag{3.5}$$

sendo:

- n, o número máximo de iteracións que o ciclo interno precisa para atopar unha nova situación estábel (o xogador entra nun ciclo inestábel cando $n = \infty$).
- T_{ii}, o tempo máximo necesario para realizar unha desas iteracións.

Entón pódese reescribir a inecuación (3.3) como

$$0 < \Delta t_f < nT_{ii} < \min(\Delta d_e) \tag{3.6}$$

Esta desigualdade representa os límites teóricos do tempo de ciclo definidos polos postulados temporais para poder garantir nun xogador ARS a reactividade e o determinismo na interpretación dos modelos. Desta expresión poden derivarse tamén os límites teóricos de cada iteración do ciclo de busca dunha situación estábel, que serán

$$0 < \Delta t_f < T_{ii} < \min(\Delta d_e)/n \tag{3.7}$$

3.3.2.4. Revisión dos postulados temporais baixo a hipótese de sincronismo forte

A proposta da aproximación síncrona á especificación de sistemas reactivos [21] e a dispoñibilidade de linguaxes como Signal [22] ou Esterel [24], motivaron unha revisión dos postulados temporais do Grafcet e da súa semántica [33][114][149]. As linguaxes síncronas baséanse na hipótese de *sincronismo forte* que establece que: '*dende un punto de vista teórico, un sistema reactivo ideal é aquel que produce as súas saídas de maneira síncrona coas súas entradas, é dicir, as súas reaccións son instantáneas (causalidade de duración nula) e deterministas*'. Dito doutra forma, un sistema reactivo ideal traballa ao ritmo que lle impón o seu contorno e, para garantir as restricción relacionadas coa seguridade, as súas reaccións teñen que ser deterministas baixo esas condicións de funcionamento.

A consideración desta hipótese ten importantes consecuencias nos postulados temporais do Grafcet [25][106]. O primeiro postulado non é compatíbel coa hipótese de sincronismo forte, pois establece que o tempo de duración dunha evolución do modelo pode ser tan pequeno como se queira mais non nulo. Obviamente isto é incompatíbel coa hipótese de causalidade de duración nula. Ademais as evolucións internas dos modelos Grafcet son síncronas (§3.3.1.4), o cal tampouco é compatíbel cos principios que fundamentan as linguaxes síncronas [128]. En canto ao o segundo postulado, establece os límites teóricos na resposta do xogador de Grafcet para garantir a reactividade do modelo. Mais, como xa foi explicado, este postulado por si só non garante ambas as dúas propiedades de interese nos sistemas reactivos (a reactividade e o determinismo), pois para garantir a segunda é preciso que o xogador poda, en todas as situacións posíbeis, evoluir a unha nova situación estábel antes da ocorrencia dun novo evento externo. Como na práctica isto non é sempre posíbel, propuxéronse dúas posíbeis interpretacións do Grafcet que priman unha das propiedades sobre a outra: a interpretación SRS prima a reactividade e a ARS o determinismo.

En consecuencia, o marco temporal do Grafcet podería ser denominado como de *sincronismo 'feble'*, pois a reactividade externa do modelo está condicionada pola súa evolución interna. Co obxecto de compatibilizar a interpretación dos modelos Grafcet coa hipótese de sincronismo forte, propúxose un novo marco temporal que establece, dende un punto de vista teórico, a independencia necesaria entre a reactividade externa do modelo e a súa evolución interna. Neste novo marco defínense dúas escalas temporais independentes, unha para a evolución interna do modelo e outra para a evolución do sistema físico controlado (percibida dende o modelo a través da variación dos valores das entradas). A fronteira conceptual entre o modelo do sistema e o sistema físico convertese así tamén nunha fronteira temporal entre a escala de tempo interna e a externa, sen relación entre elas. A definición textual é a seguinte [106]: '*a fronteira de separación entre o modelo e o sistema físico constitúe tamén unha fronteira temporal entre unha escala de tempo interna e outra externa ao modelo. Ambas as dúas escalas non teñen unha medida común.*'

Esta separación entre ambas escalas permite compatibilizar os postulados temporais do Grafcet coa hipótese de sincronismo forte. Na escala externa a resposta do modelo prodúcese instantaneamente para cada novo evento do sistema, e na escala interna o modelo evoluciona segundo unha interpretación ARS, de forma síncrona e coa duración de cada evolución infinitamente pequena mais non nula. Esta aproximación basease na suposición de que a velocidade de evolución do modelo (velocidade do sistema de control) é infinitamente rápida comparada coa velocidade do sistema controlado, ou dito doutra forma, que os intervalos de tempo internos son percibidos dende o punto de vista externo como instantáneos (Figura 3.32).

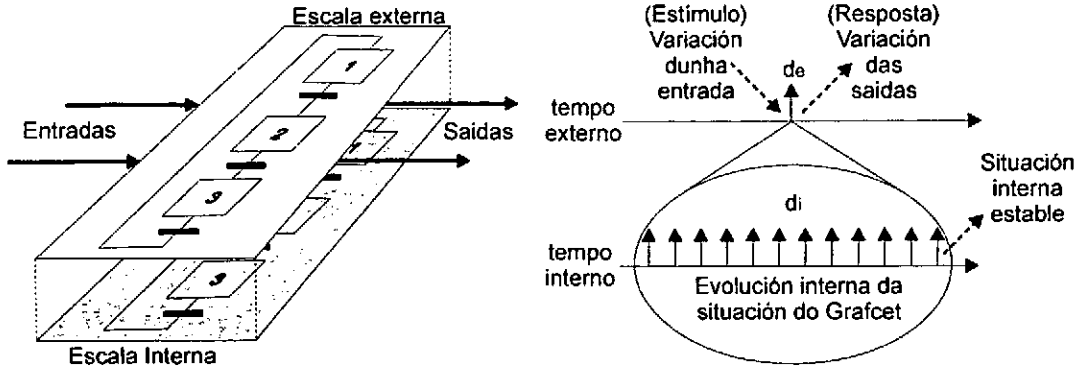


Figura 3.32. Escalas de tempo interna e externa na interpretación do Grafcet.

Os postulados temporais do Grafcet (§3.3.2.1), revisados para ter en conta as dúas escalas temporais definidas, quedan agora da maneira seguinte:

Postulado 1

Na escala de tempo externa, todo evento é tido en conta polo modelo no instante xusto da súa ocorrencia. Os cambios no estado do modelo e nos valores das saídas que provoque cada evento son percibidos nesta escala como se sucederan instantaneamente.

Postulado 2

Na escala de tempo interna, a duración dunha evolución pode ser tan pequena coma se desexe mais non pode ser nula. En consecuencia o tempo de activación dunha etapa tampouco pode ser nulo.

A consideración conxunta dambos postulados e das dúas escalas de tempo independentes definidas garanten as propiedades de reactividade e determinismo que caracterizan aos sistemas reactivos ideais. En efecto, o primeiro postulado establece que o tempo de resposta do modelo na escala de tempo externa é instantáneo, e polo tanto cumpre coa condición de causalidade nula imposta pola hipótese de sincronismo forte. Por outra banda o segundo postulado, xunto coa utilización dunha interpretación ARS dos modelos, garante simultaneamente o sincronismo da evolución e o determinismo da resposta.

As modificacións no xogador ARS que esta revisión dos postulados implican poden verse na Figura 3.33. As ecuacións que describen os límites temporais dos dous ciclos deste xogador cambian agora para ter en conta as dúas escalas de tempo independentes definidas. O tempo de ciclo interno (o ciclo que calcula a nova situación estábel despois da ocorrencia dun evento) estará medido na escala interna e dacordo ao segundo postulado virá dado por:

$$T_{ci}(\text{int}) > 0 \tag{3.8}$$

No que respecta ao tempo de ciclo externo (o ciclo que calcula o valor das variábeis de saída na nova situación estábel) medirase na escala de tempo externa e segundo o primeiro postulado virá dado por:

$$T_{ce}(\text{ext}) = 0 \tag{3.9}$$

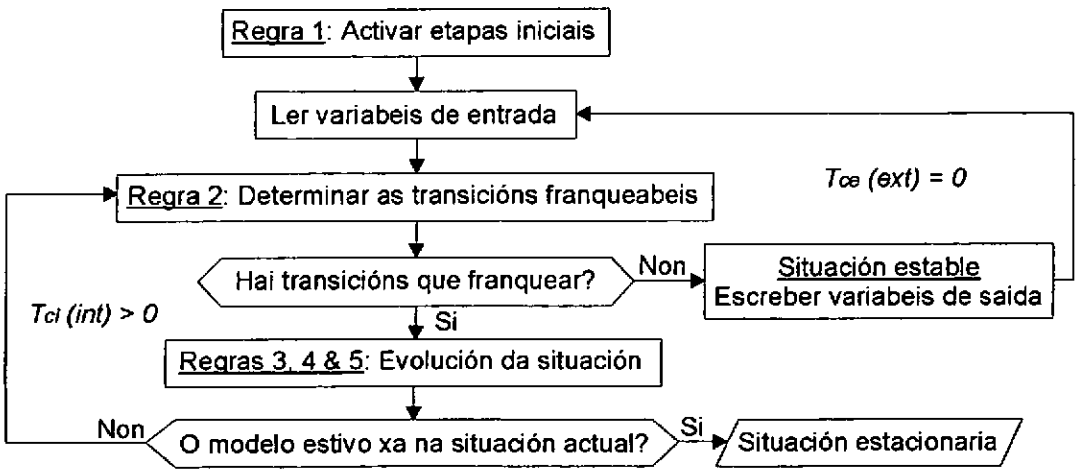


Figura 3.33. Xogador ARS con dúas escalas de tempo baixo a hipótese de sincronismo forte.

3.3.2.5. Consideración das ordes de forzado no xogador de Grafcet

As ordes de forzado (§3.2.2.4) permiten modelar relacións de dependencia entre os grafcets parciais dun modelo. Estas relacións forman unha xerarquía na que os grafcets forzados están situados en niveis inferiores ao dos grafcets 'forzadores'. Para a aplicación da xerarquía de forzado durante a interpretación dos modelos é precisa a modificación do xogador Grafcet. As ordes de forzado son ordes internas que forzan unha nova situación do modelo dende o propio modelo, polo que para evitar que entren en conflicto coas evolucións 'estructurais' obtidas como resultado da aplicación das regras de evolución, é preciso formalizar a relación entre ambas conservando a propiedade de sincronismo na evolución interna do modelo. Con este obxectivo definíronse as regras seguintes [25]:

Regra de forzado 1

As ordes de forzado son ordes internas posteriores a unha evolución estructural do modelo. Cando unha nova situación, obtida como consecuencia da aplicación das regras de evolución, contén unha ou mais ordes de forzado, os grafcets forzados pasan a estar na situación forzada de maneira inmediata e mantéñense nela mentres que as ordes que os forzan estean activas.

Regra de forzado 2

A aplicación das ordes de forzado ten prioridade diante de calquera outra actividade do modelo (evolución interna, cálculo dos novos valores das saídas, etc.).

A consecuencia directa destas dúas regras é a definición dunha orde na aplicación conxunta das ordes de forzado e as regras de evolución interna do modelo (§3.3.1). As primeiras teñen prioridade sobre as segundas, o cal implica que calquera situación obtida como consecuencia da aplicación das regras de evolución pode ser modificada antes dunha nova aplicación das regras, se algunha das etapas activas nesa situación tivera asociada unha orde de forzado. Unha nova aplicación das regras de evolución só será posíbel dende a situación obtida despois de aplicar todas as ordes de forzado asociadas ás etapas activas.

Unha consideración adicional a ter en conta durante a aplicación das ordes de forzado é a posibilidade de que se produzan *forzados en cascada*, que consisten en que algunha das etapas activadas como consecuencia da aplicación dunha orde de forzado teña asociadas á súa vez novas ordes de forzado, que terán que ser aplicadas antes de permitir unha nova evolución interna do modelo —regra de forzado 2—. Será preciso, polo tanto, garantir que durante a aplicación das ordes de forzado se mantén a coherencia entre a xerarquía de forzado e as evolucións internas do modelo.

O xogador Grafcet modificado para ter en conta as ordes de forzado pode verse na Figura 3.34. Debido a que son ordes internas, a súa aplicación realizase no ciclo interno do xogador e faise na orde definida polos niveis da xerarquía de forzado, aplicando primeiro as ordes asociadas ás etapas activas nos grafcets parciais de nivel superior e baixando na xerarquía ate que todos os niveis son cubertos. Deste xeito introdúcese, despois de cada evolución 'estructural', unha evolución asíncrona na escala de tempo interna. Sen embargo o ciclo interno considerado no seu conxunto segue conservando o seu caracter síncrono, pois a propagación das ordes de forzado é prioritaria sobre calquera outra actividade do modelo, o que implica que as regras de evolución só volverán a aplicarse unha vez que esta remate.

Outro aspecto a ter en conta durante a interpretación do modelo é o *forzado múltiple* dun mesmo grafcet parcial, que se produce cando dúas ou mais ordes de forzado forzan simultaneamente situacións diferentes nun mesmo grafcet parcial. O estándar prohibe

explicitamente esta posibilidade, polo que será preciso engadir no xogador a detección desta condición durante a propagación das ordes de forzado.

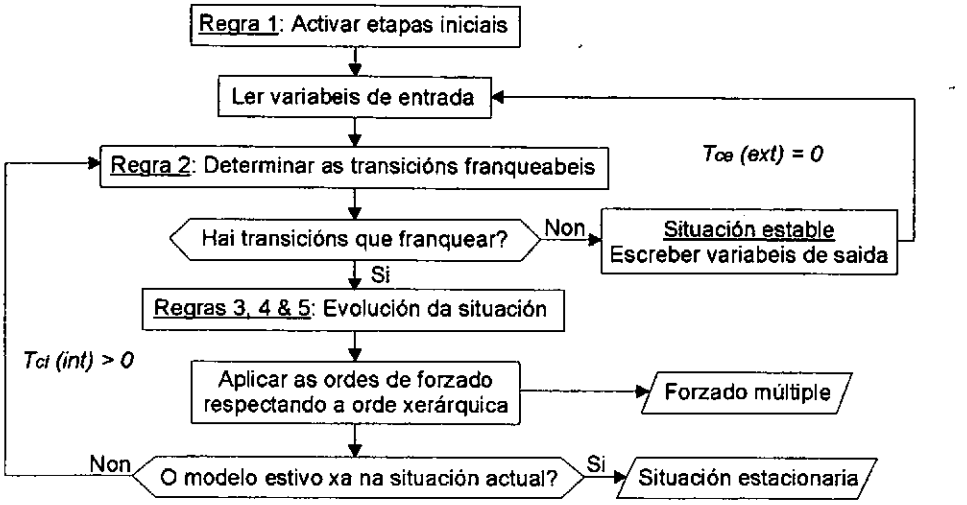


Figura 3.34. Xogador ARS de modelos Grafcet con ordes de forzado.

3.3.2.6. Consideracións sobre o uso de dúas escalas temporais independentes

Inda que a definición de dúas escalas temporais independentes permite desligar as evolucións internas do modelo da súa resposta externa e así, dende un punto de vista teórico, garantir a hipótese de sincronismo forte (§3.3.2.4), existen certas complicacións adicionais derivadas desta definición. Como as dúas escalas son independentes non poden representarse os eventos externos e internos sobre unha mesma escala de tempo, mais xa que as duracións na escala de tempo interna son infinitamente pequenas en relación á escala externa onde son percibidas como instantáneas, si que é posíbel representar o que acontece na escala interna nun instante dado da escala externa (Figura 3.35). Esta representación recebe o nome de *representación dilatada do tempo interno* e é unha propiedade da *análise non estándar* [64], utilizada na demostración formal dos postulados temporais relativos ás dúas escalas de tempo.

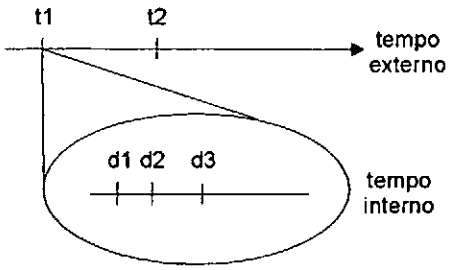


Figura 3.35. Representación dilatada do tempo interno.

As complicacións proveñen da falla dunha definición que indique como deben considerarse os eventos externos, que son flancos instantáneos no tempo externo, na escala de tempo interna, onde os instantes externos son dilatados e, teoricamente, poden ter duracións tan longas como se desexe. Como pode verse na Figura 3.36 hai dúas posíbeis respostas, que o evento sexa considerado como unha constante durante todo o período, ou ben, que a súa duración sexa nula no tempo interno e, polo tanto, poda asimilarse a un flanco ao comezo do período dilatado. A escolla dunha ou outra solución influirá na evolución interna do modelo, pois a avaliación do evento nas receptividades terá valores distintos nun e noutro caso.

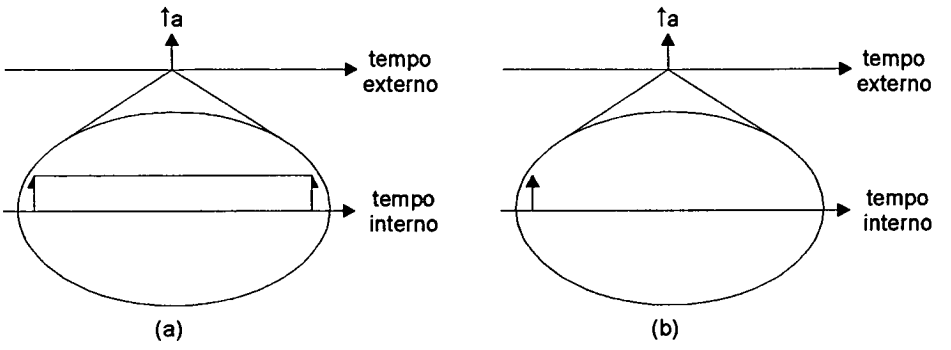


Figura 3.36. Posíbeis interpretacións dun evento externo na escala de tempo interna: (a) como unha constante; e (b) como un evento de duración nula.

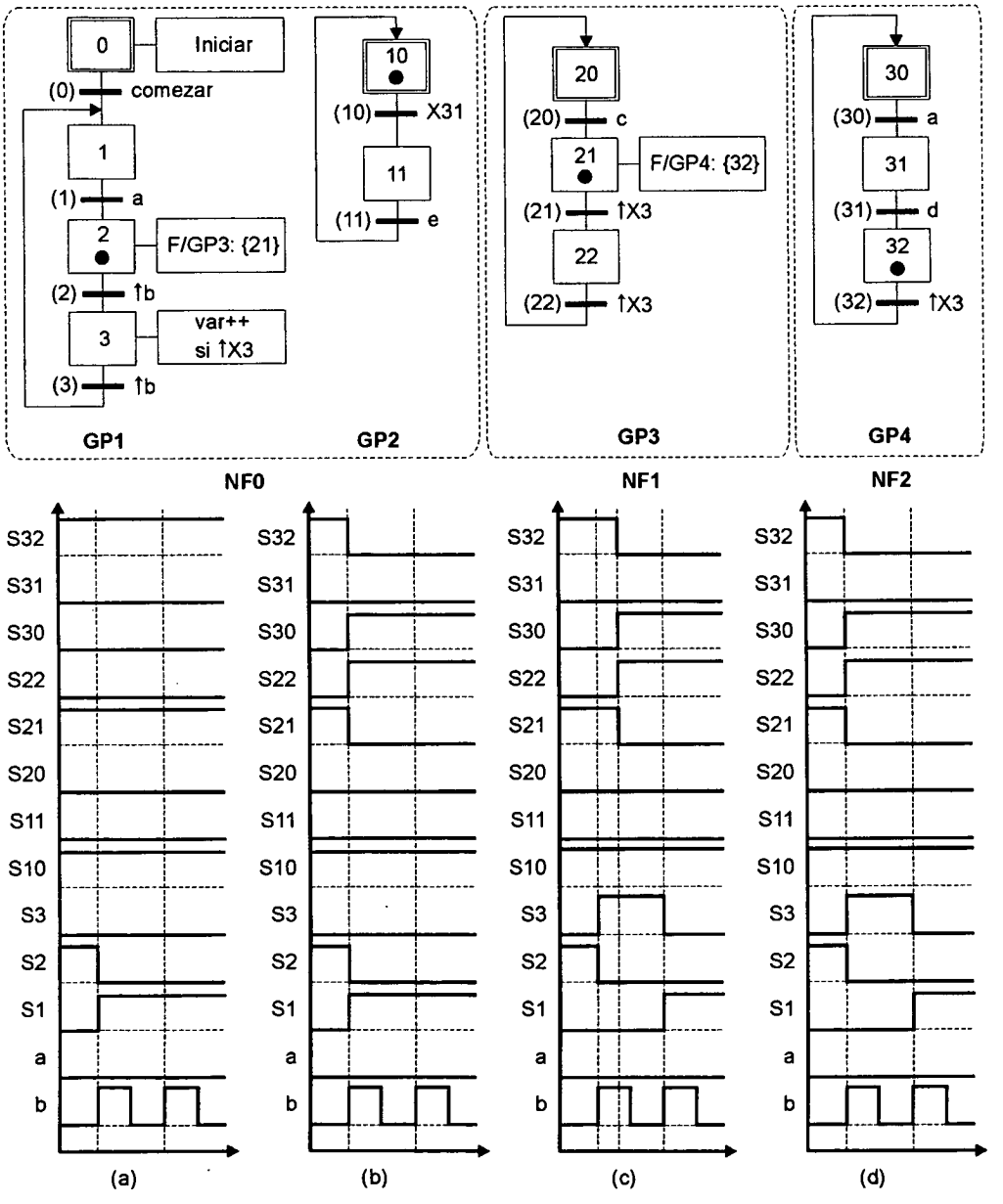


Figura 3.37. Exemplo de interpretacións dun mesmo Grafcet considerando: (a) ↑b e ↑X3 na escala externa; (b) ↑b na escala externa e ↑X3 na interna; (c) ↑b na escala interna e ↑X3 na externa; e (d) ↑b e ↑X3 na escala interna.

Ademais este problema non afecta unicamente á consideración dos eventos externos [25]. O mesmo acontece co estado de activación das etapas, cos valores das variábeis e coa consideración das accións. En efecto, durante unha evolución interna do modelo o estado de activación interno dunha etapa pode ser diferente ao seu estado de activación externo, debido a que o modelo puido mudar a súa situación interna, mais a externa só mudara cando se chegue a unha situación interna estábel. A cuestión que se suscita é cal dos dous valores utilizar durante unha evolución interna do modelo. O razoamento para as variábeis e as accións é semellante, o modelo podería incluír variábeis cuxos valores foran significativos só nunha das dúas escalas, e accións que fosen consideradas só na escala externa (nunha situación estábel) ou tamén na interna (en situacións inestábeis). A Figura 3.37 mostra un exemplo de como a interpretación dos eventos, variábeis e accións con respecto ás dúas escalas temporais modifica o comportamento externo do modelo.

3.3.3. Interpretación temporal das accións

O estándar Grafcet [164] inclúe diferentes tipos de accións que permiten o modelado de diferentes relacións temporais entre o estado de activación dunha etapa e o das variábeis modificadas nas accións que teña asociadas. Neste apartado explícanse as semánticas temporais de cada un dos tipos de accións do estándar.

3.3.3.1. Accións continuas (tipo N)

Nas *accións continuas* o estado de activación da acción coincide co da etapa (Figura 3.38). Se na especificación da acción non se indica ningún tipo, este é o utilizado por defecto.

3.3.3.2. Accións impulsionais (tipo P)

As *accións impulsionais* son accións que teñen unha duración moi pequena (Figura 3.39) e que se utilizan para xerar pulsos nas saídas. O pulso xérase no instante de activación da etapa.

3.3.3.3. Accións memorizadas (tipos S e R)

As *accións memorizadas* utilízanse cando a duración da activación dunha acción é superior ao da etapa á que está asociada. Nese caso utilízanse un par de accións S e R asociadas a etapas diferentes (Figura 3.40), na tipo S actívase a acción e na tipo R desactívase.

3.3.3.4. Accións condicionais (tipo C)

Nas *accións condicionais* o estado de activación da acción depende do estado de activación da etapa e dunha condición lóxica. A acción só estará activa cando o estea a etapa á que está asociada e a condición lóxica sexa certa (Figura 3.41).

3.3.3.5. Accións temporizadas: retardadas (tipo D) e limitadas (tipo L)

As *accións temporizadas* son un caso particular das accións condicionais nas que a condición lóxica depende do tempo de activación da etapa. Nas *accións retardadas* o instante de activación da acción é posterior ao de activación da etapa (Figura 3.42). Nas *accións limitadas* o estado de activación da acción é simultáneo co da etapa, e a súa duración está limitada, sendo sempre inferior ou igual á duración da activación da etapa (Figura 3.43).

3.3.3.6. Combinación de accións

As accións condicionais (C) e as temporizadas (D e L) poden ser utilizadas como cualificadores das accións básicas: continuas (N), impulsionais (P) e memorizadas (S, R), para formar un tipo combinado (CP, DS, SL, DC, etc.), no que a orde na que aparece cada letra é relevante na interpretación da semántica da acción (Figura 3.44; Figura 3.45; Figura 3.46).

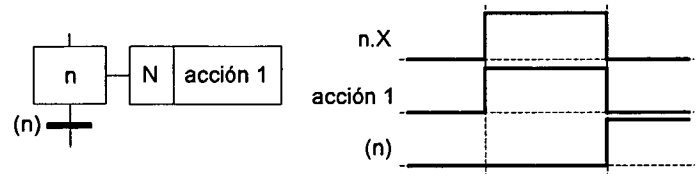


Figura 3.38. Cronograma dunha acción tipo N.

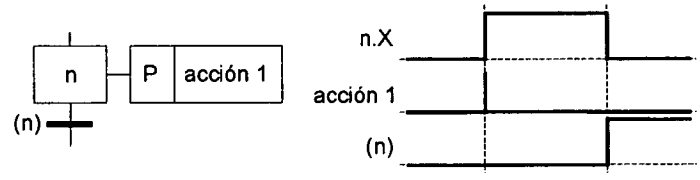


Figura 3.39. Cronograma dunha acción tipo P.

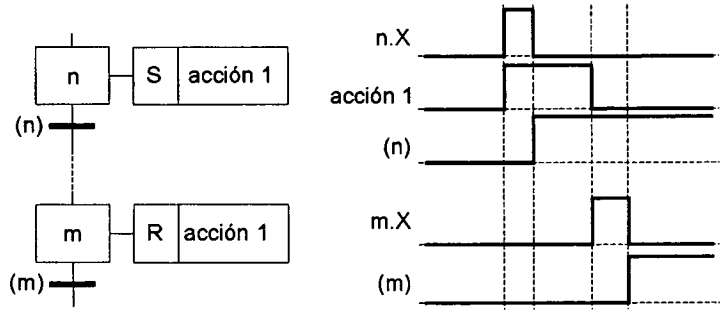


Figura 3.40. Cronograma dunha acción memorizada.

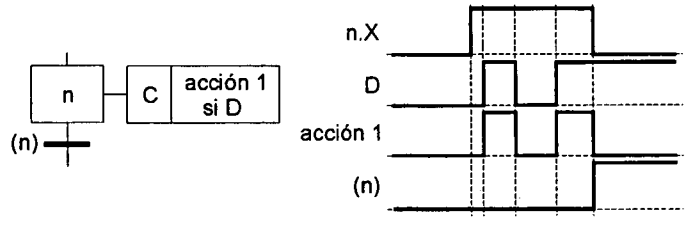


Figura 3.41. Cronograma dunha acción condicional.

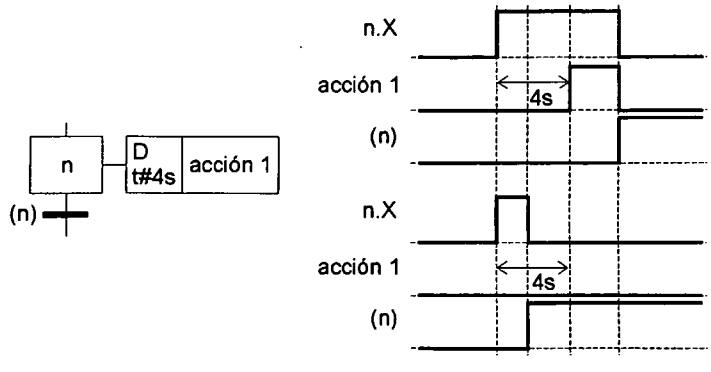


Figura 3.42. Cronograma dunha acción retardada.

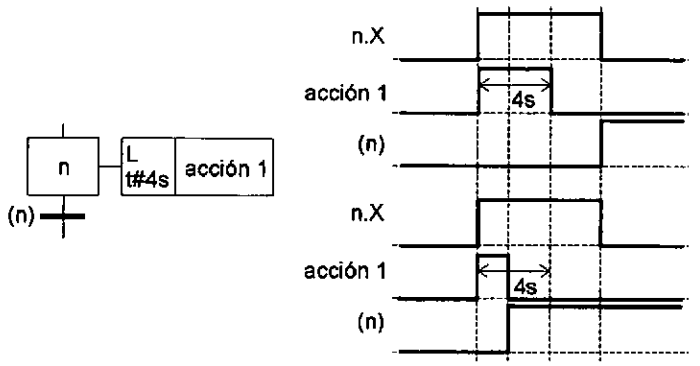


Figura 3.43. Cronograma dunha acción limitada.

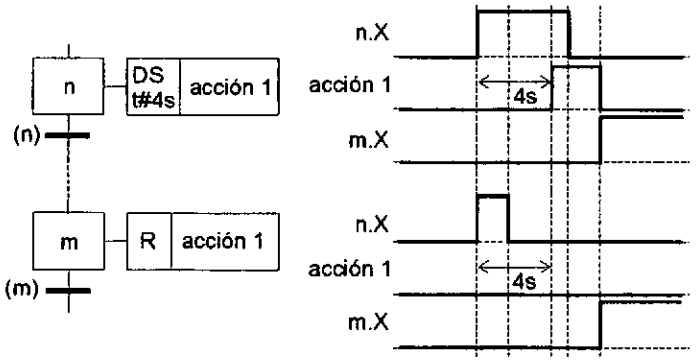


Figura 3.44. Cronograma dunha acción DS.

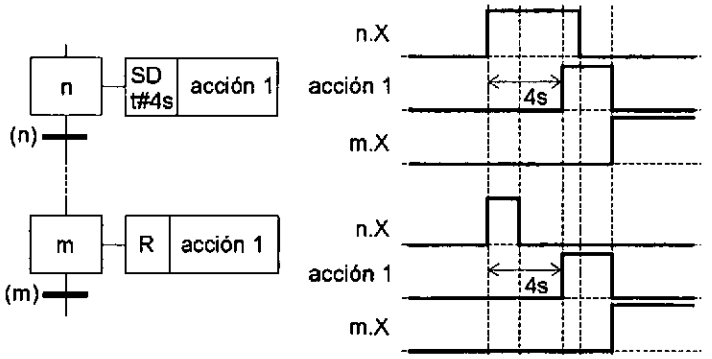


Figura 3.45. Cronograma dunha acción SD.

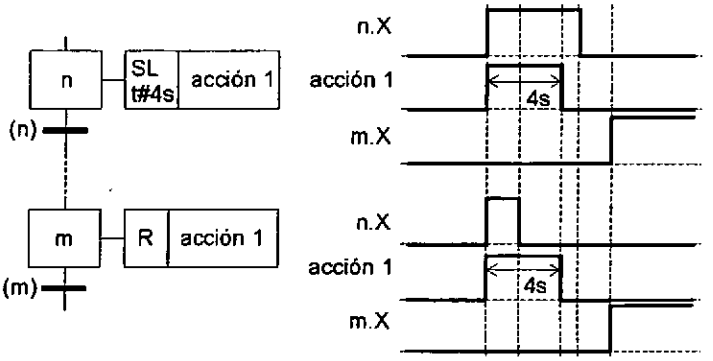


Figura 3.46. Cronograma dunha acción SL.

3.3.3.7. Funcións operativas asociadas ás accións

Como se explica en (§3.4.1) o modelo Grafcet básico formaliza unicamente dous tipos de accións: as continuas e as impulsionais[24]; e utiliza un conxunto de *funcións operativas* [30] pertencentes a un segundo nivel de descrición para modelar, desde un punto de vista externo, as diferentes semánticas temporais explicadas nos apartados previos. As funcións operativas utilizadas son as seguintes:

1. *O operador de retardo*, utilizado nas expresións lóxicas que inclúen temporizacións (por exemplo, nas accións tipos D e L). Represéntase mediante a expresión:

$$S = t1/E/t2 \tag{3.10}$$

sendo:

- *S*, a variábel booleana obtida como resultado da temporización.
- *E*, a variábel booleana utilizada como base para o cálculo da temporización.
- t_1 e t_2, os tempos de retardo aplicados a E para obter S. En concreto, os instantes de activación (t_a) e desactivación (t_d) de S calcúlanse segundo as fórmulas:

$$t_a(S) = t_a(E) + t_1 \tag{3.11}$$

$$t_d(S) = t_d(E) + t_2 \tag{3.12}$$

Cando algúns dos retardos é igual a cero pode omitirse, sendo entón E/t_2 e t_1/E as representacións utilizadas para o operador. No primeiro caso $t_1 = 0$ e $t_a(S) = t_a(E)$ e no segundo $t_2 = 0$ e $t_d(S) = t_d(E)$.

2. *O xerador de pulsos*, utilizado para xerar un pulso de duración tan pequena como se precise nas accións impulsionais (tipo P).
3. *A memoria*, utilizada para almacenar o estado de activación das accións memorizadas (tipos R e S).

3.3.3.8. Resolución de conflictos entre accións

Unha circunstancia que pode producirse durante a interpretación dos modelos Grafcet é a modificación simultánea dunha mesma variábel desde diferentes accións activas. Se as modificacións intentan asignar valores diferentes á variábel existirá unha situación de conflicto que pode dan lugar a resultados inesperados. É preciso polo tanto ou ben garantir que o modelo está libre de conflictos mediante técnicas de análise a priori, ou ben definir un método para a resolución do conflicto cando este se produza. En [114] propóñense tres posíbeis maneiras de resolver esta situación:

1. *Asignar prioridades ás etapas do modelo*, de xeito que en caso de conflicto se apliquen unicamente as accións de maior prioridade.
2. *Definir unha relación de composición* entre os valores asignados polas accións. Por exemplo, podería utilizarse o OR lóxico para os valores booleanos e a suma para os enteiros.
3. *Detectar a existencia de conflictos*, indicalo activando un sinal de alarma e deter a interpretación do modelo.

[24] Non confundir coas accións tipo P explicadas anteriormente.

Como é explicado posteriormente (§3.6.1.3), o estándar IEC 61131-3 resolve os conflictos entre asociacións definindo un bloque de control que determina o estado de activación dunha variábel ou acción cando hai varias asociacións activas que a modifican.

3.4. O modelo matemático do Grafcet

O modelo matemático aquí descrito correspóndense coa definición orixinal do Grafcet derivada da das RdP e na que se utilizan unicamente variábeis booleanas. O modelo está composto por dúas compoñentes: unha *estática*, que describe a estructura dos modelos; e outra *dinámica*, que describe as evolucións; e foi tomado das propostas aparecidas en diferentes traballos [38][100][164]. Outras aproximacións baseadas en diferentes formalizacións matemáticas do tempo e os eventos poden consultarse en [33][64][65][74][147]. Recentemente propuxéronse extensións a este modelo para mellorar as capacidades de modelado de xerarquías complexas, interrupcións ("preemption") e a aplicación do Grafcet en sistemas híbridos [75][76]. Algunhas destas propostas foron aprobadas como parte da revisión do estándar Grafcet internacional [85] e a súa incidencia na ferramenta proposta nesta tese de doutoramento pode consultarse en [133] e [134].

3.4.1. O contorno do modelo

A proposta orixinal do Grafcet propuña unha formalización baseada na suposición dunha relación entre o modelo e o seu contorno como a mostrada na Figura 3.47. Todas as entradas (i_n) e saídas (o_n) utilizadas son booleanas e as accións divídense en dúas categorías: as *continuas* e as *impulsionais*[25]. As accións continuas representan estados nos que se modifican directamente as saídas do modelo (o_2, o_4, o_6), mentres que as impulsionais representan ordes que ou ben poden ser emitidas directamente no proceso controlado (o_5*), ou ben indicar o inicio (o_8*) dunha operación auxiliar —realizada polas funcións operativas asociadas ao Grafcet (§3.3.3.7)— na que se modifique indirectamente unha saída (o_1, o_3) ou variábel interna (i_4) do modelo. As funcións operativas proporcionan diferentes posibilidades de modificación das saídas do modelo: xeración de pulsos, temporizacións e memorizacións; o que permite modelar, desde un punto de vista externo, distintos tipos de semánticas nas accións (§3.3.3): P, D, L, R, S, etc. En consecuencia no Grafcet distínguense formalmente dous niveis ou *fronteiras de descrición* [30]: o primeiro sería o formado pola definición básica, na que se dispón unicamente dos dous tipos de accións comentados, e o segundo englobaría ademais as funcións operativas asociadas que permiten o modelado nas accións de diferentes semánticas temporais desde un punto de vista externo.

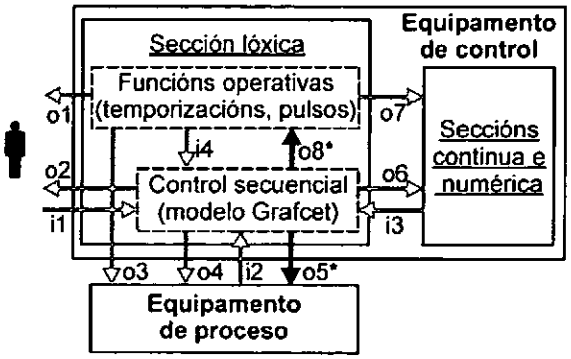


Figura 3.47. Intercambio de información entre o modelo e o seu contorno.

[25] Indicadas na Figura 3.47 cun asterisco.

3.4.2. Estructura estática

O Grafcet é un formalismo para a especificación de sistemas lóxicos secuenciais que pode ser representado como unha tupla ⟨G, Env, Int⟩, na que:

- $G = \langle S, T, L, S_0 \rangle$, é a estrutura do Grafcet. Esta estructura é un grafo bipartito dirixido, composto de dous tipos de nodos (conxuntos S e T) e un conxunto de arcos orientados (Ł):
  - $S = \{s_1, s_2, \ldots, s_m\}$, conxunto finito non baleiro de etapas. $S_0 \subset S$ é o subconxunto de etapas iniciais.
  - $T = \{t_1, t_2, \ldots, t_n\}$, conxunto finito non baleiro de transicións.
  - $L = \{l_1, l_2, \ldots, l_p\}$, conxunto finito non baleiro de arcos orientados que unen unha etapa a unha transición ou viceversa; está formado polo par $\langle L_I, L_O \rangle$ de xeito que:
    - L_I: $S \times T \rightarrow \{0, 1\}$ é a función que describe o conxunto de arcos que unen unha etapa a unha transición.
    - L_O: $T \times S \rightarrow \{0, 1\}$ é a función que describe o conxunto de arcos que unen unha transición a unha etapa.

  Utilízanse as notacións $^\circ(s_i)$ e $(s_i)^\circ$ para indicar, respectivamente, o conxunto de transicións que preceden e suceden a unha etapa. A mesma notación é utilizada para indicar o conxunto de etapas que preceden ou suceden a unha transición.

- $Env = \langle I, O, E_I, E_O \rangle$, é a interface do Grafcet co seu contorno ($I \cap O \cap E_I \cap E_O = \varnothing$):
  - $I = \{i_1, i_2, \ldots, i_q\}$, conxunto finito de entradas booleanas.
  - $O = \{o_1, o_2, \ldots, o_r\}$, conxunto finito de saídas booleanas.
  - $E_I = \{e_{i1}, e_{i2}, \ldots, e_{iu}\}$, conxunto finito de eventos de entrada.
  - $E_O = \{e_{o1}, e_{o2}, \ldots, e_{ov}\}$, conxunto finito de eventos de saída.

- $Int = \langle R, A, X \rangle$ é a interpretación do Grafcet:
  - $R = \{r_1, r_2, \ldots, r_n\}$, conxunto finito de receptividades asociadas ás transicións; $\forall t_i \in T$, r_i é a receptividade asociada á transición t_i, e está formada polo par $\langle E, C \rangle$ de xeito que:
    - $r_i.E$ é a parte da receptividade formada exclusivamente por eventos.
    - $r_i.C$ é a parte da receptividade formada exclusivamente por variábeis booleanas.

    Unha transición será franqueada cando ambas partes sexan certas e todas as etapas que a precedan estean activas.

  - $A = \{a_1, a_2, \ldots, a_w\}$, conxunto finito de accións asociadas ás etapas; $\forall s_i \in S$, $A(s_i) = \{a_{i1}, a_{i2}, \ldots, a_{ik}\}$ é o conxunto de accións asociadas á etapa s_i, e está formado polo par $\langle N, P \rangle$ de xeito que:
    - $A(s_i).N$ é o conxunto de accións continuas asociadas á etapa s_i.
    - $A(s_i).P$ é o conxunto de accións impulsionais asociadas á etapa s_i.

  - $X = \{X_i \mid (s_i \in S)\}$ é o conxunto de variábeis booleanas que almacenan o estado de activación das etapas. Relacionados con elas defínense dous conxuntos de eventos:
    - $X_a = \{X_i^a \mid (s_i \in S)\}$ é o conxunto de eventos de activación da etapa s_i.
    - $X_d = \{X_i^d \mid (s_i \in S)\}$ é o conxunto de eventos de desactivación da etapa s_i.

O conxunto $E_S = E_I \cup X_a \cup X_d$ define os eventos significativos, e dicir, os eventos que poden dar lugar a unha evolución do modelo.

3.4.3. Evolución dinámica

No modelo dinámico as regras de evolución dos modelos Grafcet (§3.3.1) son representadas mediante un conxunto de ecuacións booleanas que calculan a situación seguinte do sistema en función da situación actual e dos valores das entradas. A activación/desactivación de cada etapa do modelo é calculada mediante a seguinte ecuación:

$$X_n(t+1) = CP(s_n) \vee X_n(t) \wedge \overline{CN(s_n)} \tag{3.13}$$

sendo:

- $X_n(t)$, o estado de activación da etapa s_n no instante t
- $CP(s_n)$, a condición de activación da etapa s_n, representada como:

$$CP(s_n) = \bigvee_{i=1}^{p}\left[\left(\bigwedge_{j=1}^{m} X_j\right) \wedge r_i\right] \tag{3.14}$$

  p é o número de transicións que anteceden a s_n, e para cada transición $t_i \in \{t_1, .. , t_p\}$ que antecede á etapa:
  - *m* é o número de etapas que anteceden a t_i (e polo tanto a validan).
  - $\bigwedge_{j=1}^{m} X_j$ é a condición de validación de t_i, que será certa cando todas as etapas que anteceden a t_i estean activas.
  - r_i é a receptividade asociada a t_i.

- $CN(s_n)$ a condición de desactivación da etapa s_n, representada como:

$$CN(s_n) = \bigvee_{i=1}^{s}\left[\left(\bigwedge_{j=1}^{n} X_j\right) \wedge r_i\right] \tag{3.15}$$

  s é o número de transicións que suceden a s_n, e para cada transición $t_i \in \{t_1, .. , t_s\}$ que sucede á etapa:
  - *n* é o número de etapas que anteceden a t_i (e polo tanto a validan).
  - $\bigwedge_{j=1}^{n} X_j$ é a condición de validación de t_i, que será certa cando todas as etapas que anteceden a t_i estean activas.
  - r_i é a receptividade asociada a t_i.

Nótese que este modelo unicamente considera unha escala de tempo e non ten en conta as ordes de forzado. En [59] pode consultarse un modelo dinámico que ten en conta ambos aspectos e está baseado na representación das regras de evolución do Grafcet como ecuacións diferenciais nun grupo de Galois de orde 2.

3.4.4. Comparación con outros formalismos

Existen diferentes traballos que comparan formalmente as capacidades expresivas do Grafcet coas doutros modelos para a especificación de sistemas secuenciais e DEDS. Poden consultarse [45][46] para unha comparativa coas RdP e as máquinas de estados; e [19][27] para unha comparativa cos StateCharts. Nesta sección resúmense as principais conclusións deses traballos.

3.4.4.1. Grafcet e redes de Petri

O Grafcet é comparado cun tipo de RdP denominado RdP interpretadas (RdPI), que se caracterizan basicamente por ter operacións asociadas aos nodos e eventos e condicións asociadas ás transicións. Hai dúas diferencias fundamentais entre o Grafcet e as RdPI (supoñendo que se utilicen interfaces de E/S e interpretacións equivalentes):

1. O marcado das etapas no Grafcet é booleano (unha etapa pode estar activa ou inactiva) e nas RdPI numérico (un nodo pode conter múltiples marcas).
2. No Grafcet se hai varias transicións franqueábeis todas son simultaneamente franqueadas, mentres que nas RdPI unicamente se franquea unha e non hai unha regra definida que proporcione unha escolla determinista.

En consecuencia, para que un grafcet e unha RdPI sexan equivalentes teñen que eliminarse estas diferencias, o que se consigue impoñendo certas restricción na utilización dambos modelos. No referente ás RdPI, estas diferencias elimínanse se o modelo garante as propiedades de *seguridade*: o número de "tokens" en cada nodo da rede será menor ou igual a un en calquera marca; e *determinismo*: as condicións asociadas ás transicións simultaneamente franqueábeis serán exclusivas para evitar o franqueamento simultáneo.

Con respecto ao Grafcet, definiuse o concepto de grafcet 'san' para indicar un modelo que cumpre coas propiedades seguintes:

1. En ningún momento pode darse a circunstancia de que estando unha etapa activa, haxa unha transición franqueábel que a preceda e ningunha transición franqueábel que a suceda. A razón é que no Grafcet esta circunstancia provocaría que a etapa continuase activa — regra 5 de evolución (§3.3.1.5)—, mais nas RdPI o número de "tokens" do nodo pasaría a ser maior de un, incumpríndose a condición de seguridade.
2. Cada par de transicións que poidan ser simultaneamente franqueadas non poden ter unha etapa inmediatamente predecesora en común. Esta propiedade garante que non haberá conflictos (varias transicións cuxa activación depende do mesmo nodo) na RdPI equivalente e, polo tanto, que será determinista.
3. Ningunha etapa pode ter máis dunha transición antecesora simultaneamente disparábel. O motivo é o mesmo que na primeira propiedade, no Grafcet esta circunstancia provocaría que a etapa se activase, mentres que nas RdPI o número de "tokens" do nodo pasaría a ser igual ou maior de un.

Das consideracións anteriores dedúcese que un grafcet 'san' é equivalente a unha RdPI (cunha interface de E/S e unha interpretación equivalentes ás do Grafcet) que cumpra as propiedades de seguridade e determinismo. En [46], este tipo de redes denomínanse RdPI de comando.

3.4.4.2. Grafcet e máquinas de estados

As máquinas de estados son o modelo clásico de descrición de sistemas secuencias e divídense en dúas categorías: as *máquinas asíncronas*, nas que os cambios de estado debidos a variacións nas entradas son considerados en calquera momento de funcionamento; e as *máquinas síncronas*, nas que unicamente poden producirse en sincronía cos cambios dunha das entradas utilizada como sinal de reloxo. Nambos casos existen dous tipos de máquinas dependendo da función considerada para o cálculo das variábeis de saída: as *máquinas de Moore*, nas que os valores das saídas dependen unicamente do estado interno do sistema; e as *máquinas de Mealy*, nas que dependen tamén do valor das entradas.

Existe unha correspondencia directa entre todos os tipos de máquinas comentados e o Grafcet: os estados e transicións dunha máquina son representados, respectivamente, mediante etapas e transicións no grafcet equivalente. As diferencias entre os diferentes tipos de máquinas afectan unicamente á especificación das receptividades e accións utilizadas. Como regra xeral, a representación de máquinas síncronas require a inclusión do evento $\uparrow clk$ (sendo *clk* o sinal de reloxo) en todas as receptividades do grafcet equivalente e, independentemente do tipo de sincronismo da máquina, o cálculo das saídas nas máquinas de Moore especificarase mediante accións continuas e nas de Mealy mediante accións condicionais. Pode consultarse [46] para unha explicación detallada das diferentes posibilidades.

No que respecta ao caso contrario non sempre existe unha correspondencia directa entre o Grafcet e as máquinas de estados, e isto é debido fundamentalmente a dúas razóns:

1. No Grafcet non é preciso especificar transicións para todos os posíbeis cambios de valor nas entradas, senón unicamente para aqueles aos que o modelo é receptivo nun estado dado. En consecuencia o número de transicións utilizadas nun grafcet será, por regra xeral, menor ao da máquina de estados equivalente.
2. No Grafcet pode haber máis dunha etapa activa simultaneamente (concorrencia), o cal non pode ser representado directamente nas máquinas de estados asociando un estado a cada etapa. A forma común de obter a máquina de estados equivalente consiste en representar cun estado cada posíbel situación do grafcet (que pode estar formada por unha ou máis etapas activas simultaneamente). En xeral isto leva a obter máquinas cun número de estados moi superior ao número de etapas no grafcet equivalente (con n etapas poden representarse 2^n situacións).

Do explicado anteriormente dedúcese que a capacidade de modelado do Grafcet é equivalente á das máquinas de estados nas súas diferentes variantes, e que o número de etapas e transicións necesarios será inferior ou igual ao de estados e transicións da máquina equivalente.

3.4.4.3. Grafcet e StateCharts

Os StateCharts [78] son un formalismo baseado nas máquinas de estados para a especificación gráfica de sistemas reactivos complexos que teñen sido aplicados no modelado de sistemas "hardware", do comportamento de aplicacións "software" complexas, de sistemas de comunicacións, etc. As principais propostas deste formalismo en relación ás máquinas de estados son a representación de estructuras xerárquicas e a ortogonalidade (representación de múltiples estados independentes simultaneamente activos). É importante indicar que os StateCharts non incrementan as capacidades de modelado das máquinas de estados, senón que constitúen unha aproximación que permite reducir a explosión combinatoria e facilita o manexo de diagramas cun grande número de estados, que son habituais en sistemas complexos.

En xeral é posíbel obter unha equivalencia entre os StateCharts e o Grafcet, facendo corresponder etapas e transicións no Grafcet con estados atómicos e transicións dentro dun mesmo nivel nos StateCharts. As condicións de transición e accións teñen representación directa nambos formalismos. Cada nivel xerárquico dos StateCharts representaríase mediante un grafcet parcial no Grafcet, e as compoñentes ortogonais dos estados AND mediante grafcets conexos nos que se garanta que, como máximo, unha etapa estea activa en cada instante. Sen embargo hai características nambos formalismos que non teñen representación directa no outro, aínda que sempre é posíbel buscar unha alternativa que respecte a semántica, por exemplo:

1. As transicións entre diferentes niveis xerárquicos nos StateCharts poden representarse no Grafcet mediante transicións fonte precedendo ás etapas a activar e transicións sumidoiro sucedendo ás etapas a desactivar en cada nivel. As receptividades destas transicións relacionarán o estado de activación das etapas implicadas coas condicións de transición orixinais[26].
2. O concepto de historia dun estado nos StateCharts require da utilización de variábeis auxiliares que memoricen o estado dunha etapa e dunha aproximación semellante á explicada anteriormente para activar as últimas etapas que estiveran activas nun determinado nivel xerárquico do modelo.
3. As ordes de forzado no Grafcet poden representarse mediante transicións entre niveis xerárquicos nos StateCharts.

Do anterior dedúcese que a capacidade de modelado do Grafcet é equivalente á dos StateCharts, aínda que a representación das transicións entre niveis e o concepto de historia dun estado require, en xeral, da utilización de transicións (fonte e sumidoiro) adicionais, e de receptividades complexas que relacionen os estados de activación das etapas coas condicións de transición orixinais.

3.5. Exemplos de modelado

Neste apartado recóllense dous exemplos da aplicación do Grafcet na automatización de procesos discretos de fabricación e manipulación de pezas que son comúns a moitas instalacións industriais. Para simplificar os exemplos non se tivo en conta o control dos estados de operación do sistema nin o tratamento de condicións erróneas de funcionamento.

3.5.1. Automatización dun posto de fabricación de bridas

Neste exemplo [108] descríbese a automatización dun posto de fabricación de bridas (Figura 3.48 e Figura 3.49), que contén tamén un módulo de detección e discriminación de pezas defectuosas e outro de embalaxe. As pezas sen procesar (aneis metálicos) entran no posto pola cinta 1 e son situadas baixo a taladradora mediante a acción combinada dos cilindros 1 e 2. Cando o detector (D1) detecta a presencia dun anel, a cinta 1 parase e o cilindro 1 empurra o anel diante do cilindro 2. Este cilindro está conectado a un presostato que regula a presión á que o anel é presionado contra un tope fixo situado baixo a taladradora. Xunto a este tope hai un detector de proximidade (Dpos) que sirve para determinar se o anel está correctamente situado, pois a viruta xerada polo taladrado podería desprazalo inadvertidamente. Cando o anel está correctamente situado realizase o proceso de taladrado. A taladradora contén dous motores, un

[26] Na revisión máis recente do estándar Grafcet [85] inclúese o concepto de "enclosure steps", directamente baseado nos estados AND dos StateCharts. Con esta extensión a representación de transicións entre diferentes niveis non require máis que a modificación de receptividades para obter a mesma semántica nambos formalismos.

para o desprazamento vertical da taladradora (M2) e outro para o xiro do cabezal multibroca (M1). O arranque e parada destes motores é controlado mediante detectores de fin de carreira (Dfc1 e Dfc2 para M1 e Dfc3 e Dfc4 para M2). Unha vez rematado o proceso de taladrado o cilindro 3 retira a tapa dun burato situado baixo o anel e este cae na cinta 2 que o transporta cara ao módulo de detección de defectos e empaquetado.

O proceso de detección de pezas defectuosas realizase mediante un sistema de visión artificial que se activa cando o detector (Dva) detecta o paso dunha peza pola cinta 2. Nese intre a cinta pararase e o sistema determinará si a peza é ou non válida. En caso de non selo o cilindro 4 empurrará a peza dentro da caixa de pezas defectuosas e en caso de ser correcta continuará o funcionamento da cinta 2 que levará a peza ás caixas da cinta 3. Ao final da cinta 2 hai un detector de presencia (Dn) utilizado para contar o número de pezas que van caendo en cada caixa. Cando se enche unha caixa, a cinta 2 parase e arráncase a cinta 3 que leva unha nova caixa baixo a cinta 2. Para situar correctamente a nova caixa utilízase outro detector de presencia (Dc).

Na Táboa 3-II poden verse resumidas as entradas, saídas e contadores implicados na automatización do posto de fabricación de bridas. Na Figura 3.50 amósase o Grafcet principal de control do posto de fabricación de bridas e na Figura 3.51 os contidos da macro que controla o proceso de taladrado das pezas. Para aproveitar a posibilidade de realizar en paralelo os procesos de taladrado, discriminación de pezas defectuosas e embalaxe utilizouse no Grafcet da Figura 3.50 unha estructura semellante á da Figura 3.21, que permite alternar entre dúas secuencias de accións paralelas. Neste exemplo concreto poden realizarse en paralelo o taladrado dunha nova peza e a verificación e embalaxe doutra. Para que se procese unha nova peza é preciso que a que actualmente ocupa a taladradora pase ao subsistema de detección de fallas (etapa 2) e isto non acontecerá ate que se valide a peza anterior (etapas 8 a 12) e haxa unha caixa correctamente situada na cinta 3 (etapas 5 a 7).

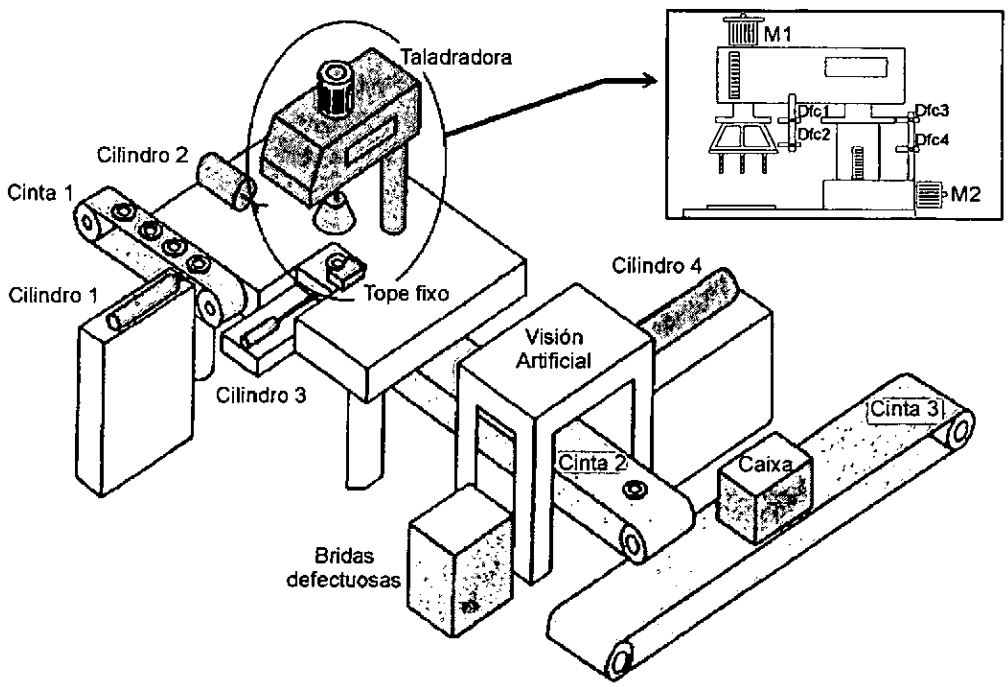


Figura 3.48. Posto de fabricación de bridas (vista 1).

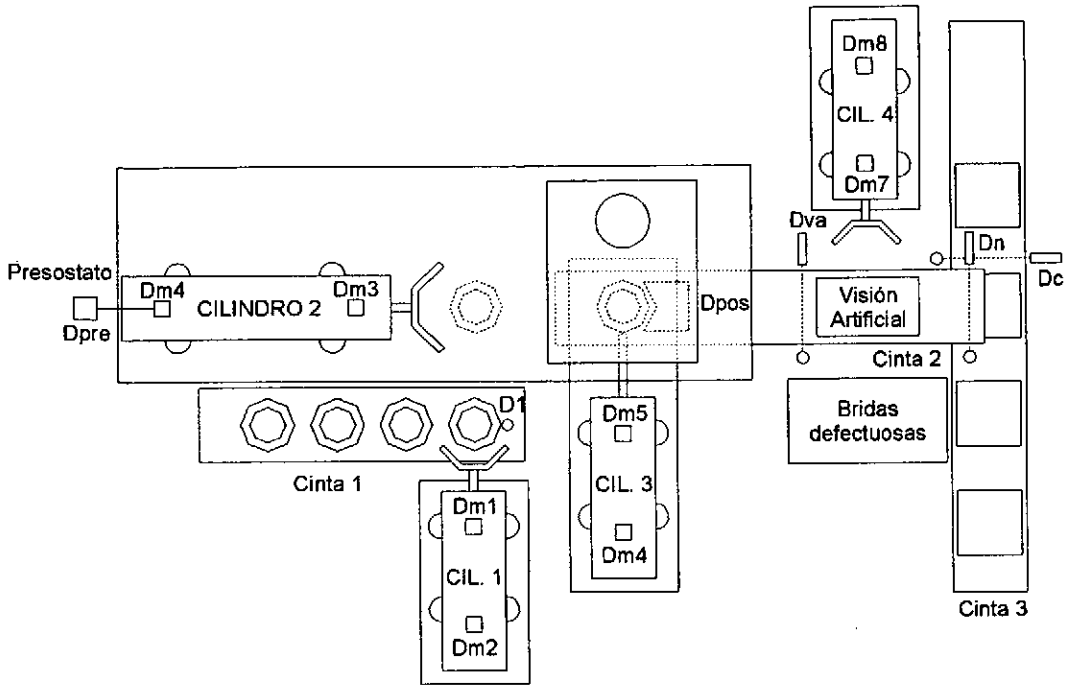


Figura 3.49. Posto de fabricación de bridas (vista 2).

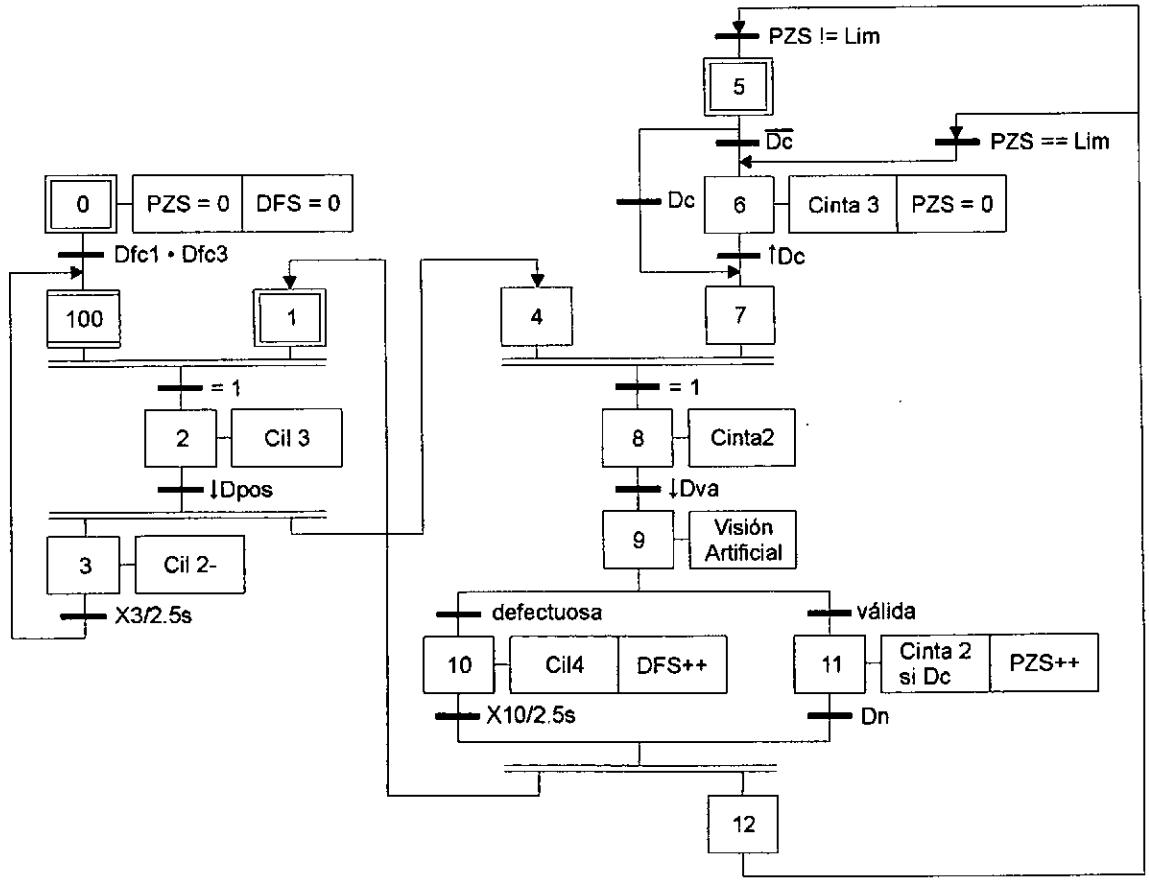


Figura 3.50. Grafcet principal de control do posto de fabricación de bridas.

Entradas		Saídas		Contadores	
D1	Detector de proximidade da cinta 1	Cil1	Avance do cilindro 1	PZS	Pezas na caixa actual
Dpre	Presostato do cilindro 2	Cil2+	Avance do cilindro 2	DFS	Total pezas defectuosas
Dpos	Detector de proximidade do tope fixo	Cil2-	Retroceso do cilindro 2		
Dfc1	Detector superior de final de carreira do motor 1	Cil3	Avance do cilindro 3		
Dfc2	Detector inferior de final de carreira do motor 1	Cil4	Avance do cilindro 4		
Dfc3	Detector superior de final de carreira do motor 2	Cinta1	Arranque da cinta 1		
Dfc4	Detector inferior de final de carreira do motor 2	Cinta2	Arranque da cinta 2		
Dva	Detector de presencia na entrada ao módulo de VA	Cinta3	Arranque da cinta 3		
Valida	Indicador de peza válida	M1+	Avance do motor 1		
Defectuosa	Indicador de peza defectuosa	M1-	Retroceso do motor 1		
Dn	Detector de presencia ao final da cinta 2	M2+	Avance do motor 2		
Dc	Detector de caixa na cinta 3	M2-	Retroceso do motor 2		
Lim	Número de pezas por caixa				

Táboa 3-II. Entradas, saídas e contadores utilizados na automatización do posto de fabricación de bridas.

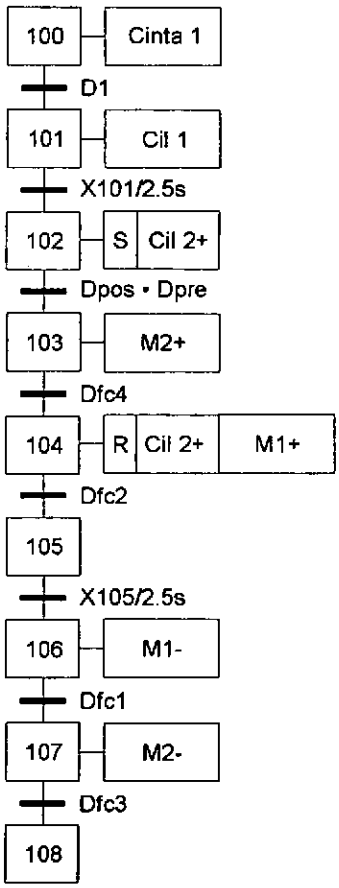


Figura 3.51. Contidos da macro 100 que controla o proceso de taladrado dunha peza.

3.5.2. Automatización dunha cela de fabricación flexible

O problema a resolver neste exemplo [30] consiste na automatización dunha cela de fabricación flexíbel como a que pode verse na Figura 3.52. Esta cela está composta por unha cinta de transporte pola que circulan os palés que conteñen as pezas a ser procesadas, dúas máquinas (unha fresadora e un torno), tres postos (PE, PS e PT) para o desprazamento das pezas dende as máquinas aos palés ou viceversa, dous robots que realizan eses desprazamentos,

un punto de entrada na cela das pezas a procesar e outro de saída da cela das pezas xa procesadas. Cada peza que entra na cela leva asociado un código que indica as operacións que xa se realizaron sobre ela e as que quedan por realizar. Este código pode ter un dos valores indicados na Táboa 3-III.

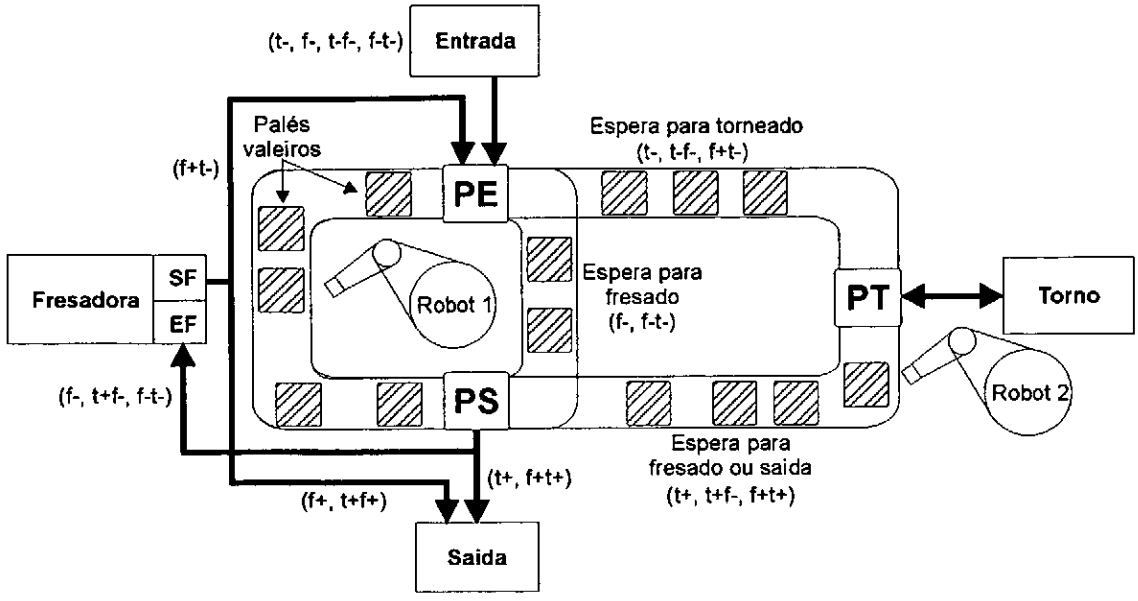


Figura 3.52. Cela de fabricación flexíbel.

t*	Peza para torneado
f*	Peza para fresado
t*f*	Peza para torneado e fresado (nesa orde)
f*t*	Peza para fresado e torneado (nesa orde)
* =[+|–] indica se a operación xa foi realizada ou aínda hai que facela.	

Táboa 3-III. Valores do código de cada peza.

Os robots encárganse das operacións de movemento das pezas desde as máquinas e puntos de entrada e saída da cela cara aos palés situados nos postos para a transferencia de pezas (PE, PS e PT). Así o robot 2 move as pezas entre o palé situado no posto PT e o torno, e o robot 1 pode realizar algún dos movementos indicados na Táboa 3-IV.

E⇒PE	Do punto de entrada ao palé baleiro situado no posto PE (pezas novas)
PS⇒S	Do palé situado no posto PS ao punto de saída (pezas rematadas)
PS⇒EF	Do palé situado no posto PS á entrada da fresadora
SF⇒PE	Da saída da fresadora ao palé baleiro situado no posto PE
SF⇒S	Da saída da fresadora ao punto de saída (pezas rematadas)

Táboa 3-IV. Movementos de pezas que pode realizar o robot 1.

A efectos do exemplo o mais interesante é o control do robot1 polo que se obviaron as operacións de fresado, torneado, o control do robot 2, a carga e descarga de palés nos puntos PE, PS e PT e a descrición detallada da secuencia de movemento dunha peza dun posto a outro. Algunha destas operacións aparecen no grafcet de control do robot 1 representadas mediante macroetapas, mais no exemplo non se inclúen os contidos das macroexpansións. As entradas e

variábeis utilizadas son as indicadas na Táboa 3-V e o grafcet principal de control do robot 1 é o amosado na Figura 3.53. O estado de activación dalgunhas das etapas deste grafcet teñen un significado específico que se recolle na Táboa 3-VI.

Entradas		Variábeis	
On	Sinal de funcionamento da cela	X	Orixe da operación de movemento dunha peza
Peza	Presencia dunha peza na entrada	Y	Final da operación de movemento dunha peza
Fin_op	Final da operación en curso		
Saida	Indicador de saída dunha peza		
Fe	Indicador de entrada dunha peza na fresadora		

Táboa 3-V. Entradas e variábeis utilizadas na automatización da cela de fabricación flexíbel.

Etapa	Significado cando está activa	Etapa	Significado cando está activa
2	Peza agardando na entrada a ser movida a PE	9	EF libre
3	Robot 1 libre	10	Peza en PS agardando a ser movida a EF
4	Palé baleiro no posto PE	11	Peza en PS agardando a ser movida á saída
5	Peza en SF agardando a ser movida a PE	12	Peza agardando a ser retirada da saída
6	Saída da fresadora libre	100	Movemento dunha peza dun punto a outro
7	Peza en SF agardando a ser movida á saída	200	Carga e descarga de palés no posto PE
8	Peza en EF	300	Carga e descarga de palés no posto PS

Táboa 3-VI. Significado do estado de activación das etapas do Grafcet do robot 1.

O grafcet de control do robot 1 debe garantir que unicamente unha das cinco posíbeis secuencias de control para o movemento de pezas está activa. Isto conséguese pola propia estructura do grafcet e mediante as condicións asociadas ás transicións 2, 4, 7, 9 e 11. Estas condicións impoñen unhas prioridades nos posíbeis movementos do robot, sendo mais prioritarios os movementos que sacan pezas da cela cara ao posto de saída e menos prioritarios os que introducen pezas dende a entrada cara ao posto PE, co obxectivo de evitar a saturación do sistema. A orde de prioridades dos movementos é SF⇒S (transición 7), PS⇒S (transición 11), PS⇒EF (transición 9), SF⇒PE (transición 4) e E⇒PE (transición 2).

Como pode comprobarse no grafcet da Figura 3.53, para que as transicións que dan paso ás secuencias de control dos movementos do robot estean validadas, o robot debe estar inactivo —etapa 3 activa—. Ademais a transición 7 (a máis prioritaria) estará validada unicamente cando haxa unha peza agardando na saída da fresadora para ser movida á saída da cela —etapa 7 activa— e só será franqueada cando a saída non estea ocupada —etapa 12 non activa—. Esta transición ten maior prioridade que as transicións 9 e 11, xa que estas transicións só poden ser franqueadas cando non haxa pezas agardando na saída da fresadora —etapa 7 non activa—. Ademais as transicións 9 e 11 non poden estar simultaneamente validadas pois as etapas 10 e 11, que representan a presencia dunha peza agardando no posto PS, non poden estar activas simultaneamente. Como estas transicións son as que dan paso ás secuencias de movemento dunha peza dende o posto PS, só unha delas vai estar validada cando haxa unha peza en PS.

Algo semellante acontece coas transicións 4 e 7 que controlan as secuencias de movemento de pezas dende a saída da fresadora. Para que a transición 4 estea validada ten que estar activa a etapa 5, e para que a transición 7 estea validada ten que estar activa a etapa 7. Ambas as dúas etapas —5 e 7— representan a presencia dunha peza agardando na saída da fresadora e só unha das dúas pode estar activa simultaneamente. Ademais para que a transición 4 sexa franqueada non pode haber ningunha peza agardando en PS —macro 300 activa—, o que garante que as

transicións 9 e 11 (que son as que controlan o movemento de pezas dende PS) van ter prioridade sobre a transición 4.

A transición con menor prioridade é a transición 2, que da paso á secuencia de control para o movemento de pezas dende a entrada da cela ata o posto PE. Esta transición só vai estar validada cando haxa unha peza agardando na entrada —etapa 2 activa—, o robot 1 estea libre —etapa 3 activa— e haxa un palé baleiro no posto PE —etapa 4 activa—. Unha vez que se cumpran esas condicións, para poder franquear a transición ten que cumprirse ademais que non haxa pezas agardando na saída da fresadora —etapa 6 activa— ou no posto PS —macro 300 activa—. Isto garante que as transicións 4 e 7 e as transicións 9 e 11, respectivamente, van ser máis prioritarias que a transición 2.

A secuencia que sigue unha peza dende que chega á entrada da cela ate que sae dela é a seguinte:

- Cando chega a peza á cela (Peza = *true*), se esta está en funcionamento (ON = *true*), actívase a etapa 2.
- Unha vez que o robot queda libre —etapa 3 activa—, haxa un palé baleiro no posto PE —etapa 4 activa— e non haxa pezas agardando nin na saída da fresadora nin no posto PS —etapas 6 e 300 activas— iníciase a secuencia de movemento da peza dende a entrada ate o posto PE —secuencia 50, 100—. Ao remate desta secuencia actívanse as etapas 1 (para aceptar novas pezas na entrada da cela), 3 (robot libre) e 200 (macro que despraza o palé ocupado cara á zona de espera correcta dacordo ao código da peza e introduce un novo palé baleiro no posto PE).
- Ao rematar a macro 200, actívase a etapa 4 para indicar que un novo palé baleiro está na posición PE. A peza estará agora sobre un palé na zona de espera do torno se o seu código é *t-* ou *t-f-*, ou na zona de espera do posto PS se o seu código é *f-* ou *f-t-*. No primeiro dos casos a peza, despois de ser torneada, é posta de novo sobre un palé na zona de espera do posto PS e o seu código pasará a ser *t+* ou *t+f-*. En calquera caso o palé no que está a peza será cargado no posto PS na macro 300. Se xa se remataron todos os procesos sobre ela (código *t+*) actívase a etapa 11, que indica que a peza está en PS agardando a ser movida á saída da cela. Se aínda hai que fresala (códigos *f-*, *f-t-* ou *t+f-*) activarase a etapa 10 que indica que hai unha peza en PS agardando a ser movida á entrada da fresadora.
- Se a peza está agardando en PS a ser movida á saída —etapa 11 activa—, cando o robot estea libre —etapa 3 activa—, non haxa outra peza agradando na saída da fresadora —etapa 7 inactiva— e a saída estea libre —etapa 12 inactiva— iniciarase a secuencia 90, 100, que move a peza dende o posto PS á saída. Cando esta secuencia remate liberarase o robot —actívase a etapa 3—, retirarase o palé baleiro do posto PS —na macro 300— para cargar un novo palé, e activarase a etapa 12 que indica que hai unha peza na saída da cela agardando a ser retirada.
- Se a peza está agardando en PS a ser movida á fresadora —etapa 10 activa—, cando o robot e a entrada da fresadora queden libres —etapas 3 e 9 activas—, e non haxa outra peza agardando na saída da fresadora —etapa 7 inactiva—, iniciarase a secuencia 80, 100 que move a peza dende o posto PS á entrada da fresadora (EF). Cando esta secuencia remate, liberarase o robot —activase a etapa 3—, retirarase o palé baleiro do posto PS —na macro 300— para cargar un novo palé, e activarase a etapa 8 que indica que hai unha peza agardando na entrada da fresadora.

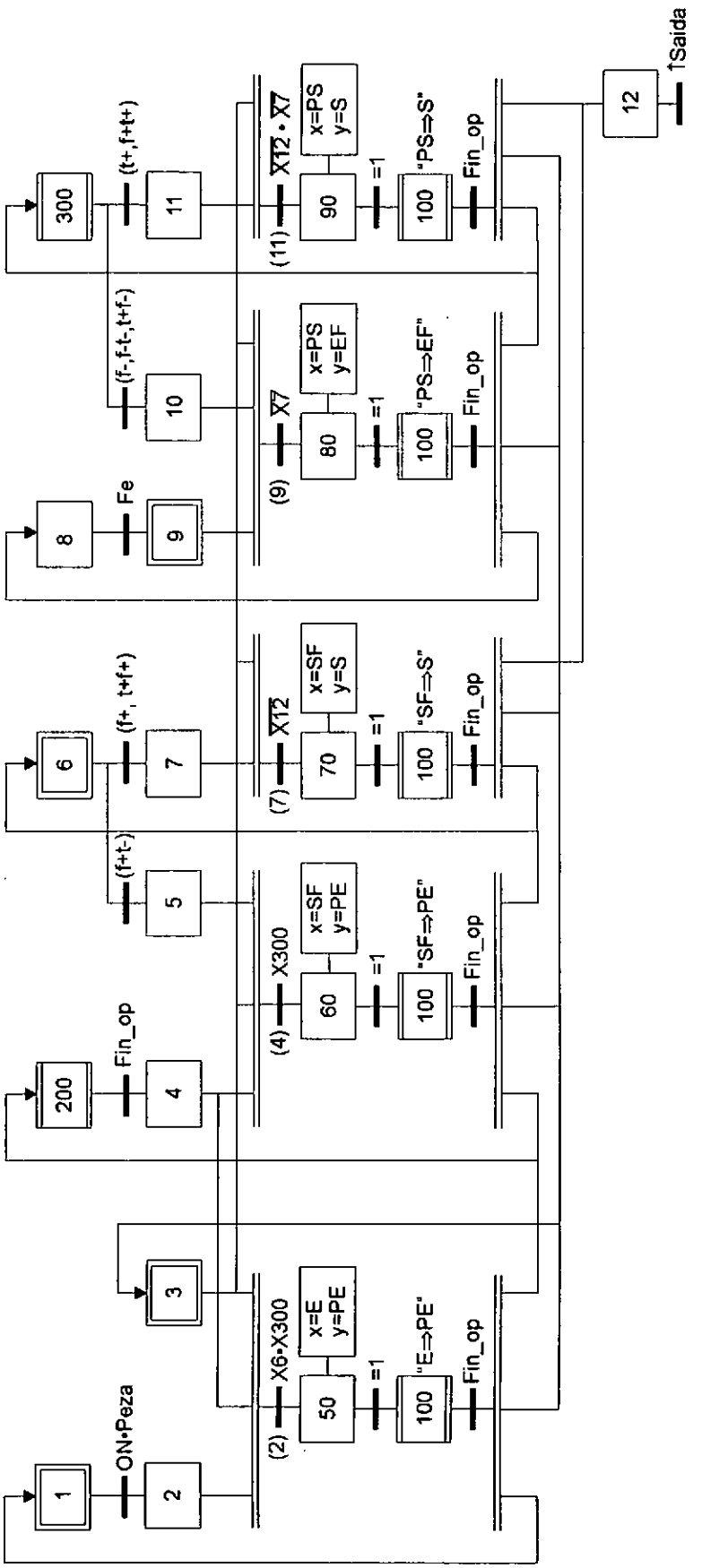


Figura 3.53. Grafcet principal de control do robot 1.

- As etapas 5 e 7 actívanse cando remata a operación de fresado sobre unha peza e esta é depositada na saída da fresadora. A etapa 5 actívase cando á peza aínda lle queda pasar polo torno (código *f+t-*), o que indica que a peza agarda a ser movida a un palé baleiro no posto PE para ser posteriormente transportada ao torno. En canto á etapa 7, actívase cando xa se remataron todas as operacións sobre a peza (códigos *f+* ou *t+f+*) e indica que a peza está agardando a ser movida á saída da cela. Este é o caso prioritario e abondará con que o robot e mais a saída da cela estean libres —etapa 3 activa e 12 inactiva, respectivamente— para que se inicie a secuencia 70, 100 que moverá a peza dende a saída da fresadora ate a saída da cela. Unha vez rematada a secuencia, liberase o robot —actívase a etapa 3—, indícase que a saída da fresadora está libre —etapa 6 activa— e que na saída da cela hai unha peza agardando a ser retirada —etapa 12 activa—.
- No caso en que a peza está agardando a ser movida dende a saída da fresadora ate o posto PE —etapa 5 activa—, cando o robot quede libre —etapa 3 activa—, haxa un palé baleiro situado no posto PE —etapa 4 activa— e non haxa ningunha peza agardando no posto PS a ser movida —macro 300 activa—, iniciarase a secuencia 60, 100 para realizar dito movemento. Ao rematar a secuencia liberarase o robot —etapa 3 activa—, activarase a macro 200 para dirixir o palé sobre o que está agora a peza cara á zona de espera para o torno, cargarase un novo palé baleiro no posto PE e activarase a etapa 6 que indica que a saída da fresadora está agora libre.

3.6. O estándar IEC 61131-3 e o SFC

O Grafcet foi inicialmente concibido como un estándar gráfico (IEC 60848, [84]) para a especificación de controladores lóxicos secuenciais independente da tecnoloxía utilizada na implementación: pneumática, eléctrica, microprocesadores, PLCs, etc. O equipamento máis utilizado nos sistemas industriais para a implementación de controladores lóxicos é o PLC. As primeiras linguaxes de programación destes equipos estaban baseadas na representación das relacións lóxicas de E/S mediante a utilización de listas de instruccións ou diagramas de contactos. Con estas linguaxes a programación de secuencias de control complexas resulta difícil, pois non proporcionan soporte á estructuración e non é fácil abstraer a estructura de control a partir do código dos programas. Os fabricantes de PLCs decatáronse pronto de que o Grafcet proporcionaba unha linguaxe gráfica, fácil de entender e utilizar, coa que a programación e depuración de secuencias complexas resultaba moito máis simple, polo que comezaron a incluílo como linguaxe de programación nos seus ambientes de desenvolvemento. Sen embargo a carencia dunha especificación que indicase de maneira unívoca como implementar o Grafcet nesta nova función levou a unha situación de coexistencia de diferentes dialectos incompatíbeis entre si, que dependían en boa medida das capacidades proporcionadas polo "hardware" dos fabricantes. Esta situación tiña numerosas desvantaxes [105]:

- *A dependencia dun fabricante*, pois estes ofrecían solucións baseadas en arquitecturas propietarias que dificilmente podían integrarse.
- *Os custos de mantemento*, a dependencia dun fabricante provoca que non sexa posíbel dispor de equipamentos equivalentes proporcionados por terceiros. Esta ausencia de competencia provoca que os prezos dos recambios e actualizacións sexa maior.
- *Os custos de formación*, a ausencia dunha interface común dificulta a formación de operadores e enxeñeiros no manexo do "hardware" dos diferentes fabricantes.
- *A dificultade na adopción de solucións heteroxéneas*, debido á complexidade inherente á integración de equipos de diferentes fabricantes.

O estándar IEC 61131-3 [86] xurde como un intento de unificar as diferentes linguaxes utilizadas para a programación de PLCs co obxectivo de permitir a aparición de ferramentas abertas que faciliten a portabilidade e permitan reducir a dependencia dos fabricantes existente ata ese momento. Unha das linguaxes incluída no estándar é o SFC, unha versión do Grafcet adaptada á programación de PLCs. Neste apartado resúmense as características máis relevantes do SFC, especialmente aquelas que presentan diferencias coa definición do Grafcet vista anteriormente. O estándar IEC 61131-3 resume estas características mediante táboas, que poden ser consultadas no Anexo A.

3.6.1. O SFC

O SFC é utilizado para a estructuración da secuencia de control interna nos diferentes tipos de POUs (funcións, bloques función e programas) definidos no modelo "software" proposto como parte do estándar IEC 61131-3. A súa definición está baseada no estándar IEC 60848 coas modificacións precisas para converter o que é un estándar de especificación e documentación nunha linguaxe de programación.

3.6.1.1. Etapas

Adóptase a representación gráfica definida polo estándar IEC 60848 para as etapas e etapas iniciais (§3.2.1.1). Tamén se definen dúas variábeis que estarán asociadas a cada etapa:

- *Un "flag" booleano* que almacena o estado da etapa, o seu valor inicial será 0 excepto nas etapas iniciais que será 1.
- *Unha variábel (tipo TIME)* que almacena o tempo transcorrido desde a última activación da etapa, o seu valor inicial é t#0s. Esta variábel almacena a información mesmo despois da desactivación da etapa e non se reinicia ata que a etapa volve activarse.

  Ambas variábeis son só de lectura, e o estándar indica que debe considerarse un erro calquera intento de modificar o seu valor.

Un aspecto no que difiren o SFC e o Grafcet é a restricción de permitir unicamente unha etapa inicial en cada *rede SFC* —que é o equivalente a un grafcet conexo (§3.2.2.3)—.

3.6.1.2. Transicións

Adóptase a representación gráfica definida polo estándar IEC 60848 para as transicións (§3.2.1.2). As condicións de transición poden especificarse utilizando as linguaxes ST, LD ou FBD directamente no SFC, ou ben definindo un bloque de código con calquera das linguaxes do estándar: LD, FBD, IL ou ST e asociándoo á transición mediante o seu nome. O estándar especifica que calquera 'efecto colateral' (modificación dunha variábel) provocado pola avaliación dunha condición de transición será considerado coma un erro.

3.6.1.3. Accións

Adóptase a representación gráfica definida polo estándar IEC 60848 para as accións (§3.2.1.5). Cada etapa pode ter asociadas 0 ou mais accións, unha etapa sen accións asociadas equivale a unha función WAIT que agarda a que algunha das transicións que suceden á etapa sexa franqueada. A definición de cada acción faise nun bloque de definición de acción ("action block"), que é utilizado como un elemento gráfico máis nos diagramas LD e FBD e que se

corresponde coa especificación detallada definida no estándar IEC 60848 (Figura 3.5) e está formada por 4 campos:

1. *Campo 'a'*, o tipo da acción.
2. *Campo 'b'*, o nome da acción.
3. *Campo 'c'*, o valor booleano utilizado para indicar a finalización da acción, condicións de erro, etc.
4. *Campo 'd'*, o código da definición da acción, no que pode utilizarse calquera das outras linguaxes definidas polo estándar, incluído o SFC.

O estándar IEC 61131-3 é mais específico que o IEC 60848 na definición dos tipos de accións e a súa semántica temporal. Ademais establece que as accións temporizadas (tipos L, D, SD, DS e SL) terán asociada unha variábel tipo TIME na que se almacene a duración dende a activación da acción.

Encanto á resolución de conflictos entre accións (§3.3.3.8), o estándar IEC 61131-3 define un bloque de control de accións ("ACTION_CONTROL function block", Figura 3.54) de uso interno (é dicir, non accesíbel ao usuario) que define a semántica de execución das accións cando son activadas simultaneamente desde diferentes etapas. O estándar establece que toda acción deberá ter asociada unha instancia deste tipo de bloque, describe a súa estructura interna (Figura 3.55) e indica as regras de conexión entre as accións e os bloques que as controlan. Unha acción será executada de maneira continua mentres a saída Q do bloque que a controla sexa certa, e unha vez máis despois dun flanco negativo de Q. A Figura 3.56 mostra un exemplo da utilización dos bloques de control de acción.

A execución da acción unha vez máis inmediatamente despois dun flanco descendente da saída Q, aspecto que foi introducido no estándar para permitir que unha acción poida reiniciar as variábeis que utiliza, ten creado certa confusión na interpretación temporal das accións tipo P, pois con esta especificación son executadas dúas veces: unha no flanco de subida da saída Q e outra no flanco de baixada. Este comportamento é pouco intuitivo polo que se propuxeron dous novos tipos de accións a incorporar nas futuras ampliacións do estándar: as tipo P1 e P0, que son executadas respectivamente nos instantes de activación e desactivación de Q.

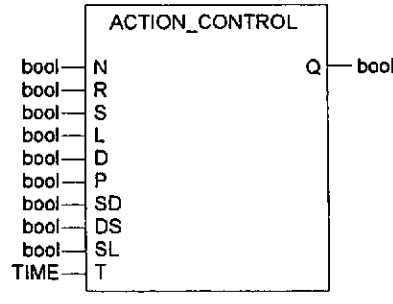


Figura 3.54. Interface externa dun bloque de control de accións.

3.6.1.4. Estructuras de control

O estándar IEC 61131-3 adopta as mesmas estructuras de control definidas no IEC 60848 (§3.2.3): secuencia simple, selección de secuencia, secuencias múltiples e saltos de secuencia. Sen embargo non inclúe as macroetapas (§3.2.2.2) e permite especificar a prioridade na avaliación das condicións de transición co fin de evitar as situacións de paralelismo interpretado (§3.2.3.10).

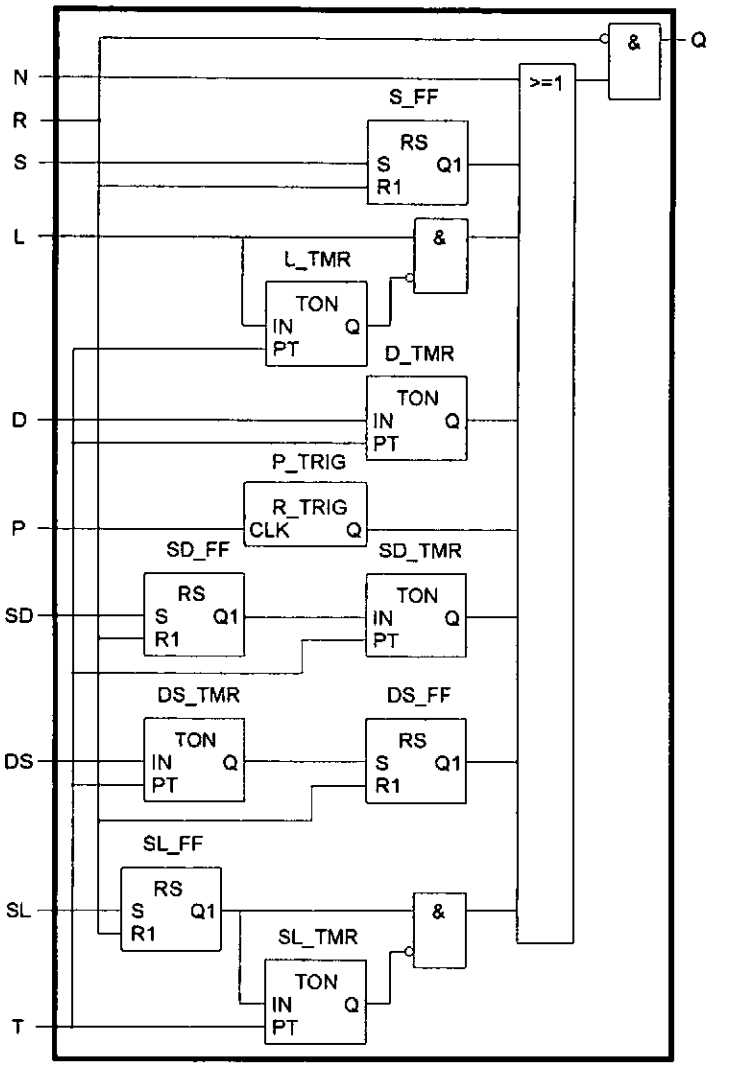


Figura 3.55. Estructura interna dun bloque de control de accións.

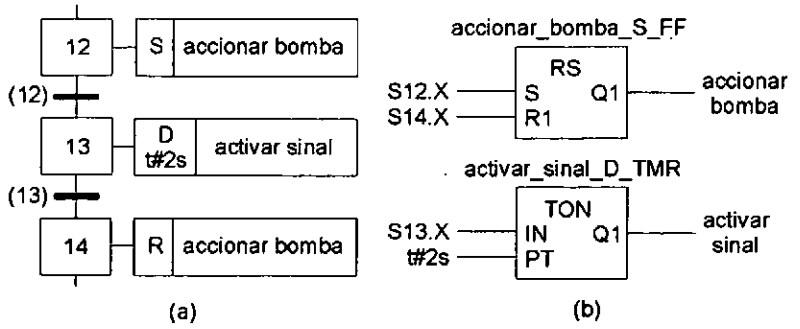


Figura 3.56. Exemplo do uso dos bloques de control de accións: (a) SFC con distintos tipos de accións; e (b) bloques de control de acción equivalentes.

3.6.1.5. Regras de evolución

As regras de evolución (§3.3.1) adoptadas polo IEC 61131-3 son as do IEC 60848 con pequenas variacións:

1. Só pode haber unha etapa inicial en cada rede SFC, polo que a situación inicial virá determinada polo estado desa etapa.

2. As receptividades asociadas ás transicións que suceden a unha etapa non son avaliadas ata que os efectos da activación da etapa sexan propagados na POU á que a etapa pertenza.
3. A regra sobre a activación/desactivación simultánea dunha etapa non se inclúe no SFC (§3.3.1.5), non obstante o estándar especifica que a definición de SFCs inseguros (Figura 3.57.a) que poidan dar lugar a situacións de paralelismo interpretado (§3.2.3.10), así como a definición de SFCs que conteñan etapas inalcanzábeis (Figura 3.57.b) deben considerarse como erros.

Ademais adóptase para o SFC o postulado temporal do Grafcet (§3.3.2.1) que establece que o tempo de franqueamento dunha transición (e polo tanto tamén o de activación dunha etapa) poderá ser tan pequeno como se queira, mais non nulo. Isto implica que o estándar IEC 61131-3 adopta para o SFC unha semántica SRS (§3.3.2.2).

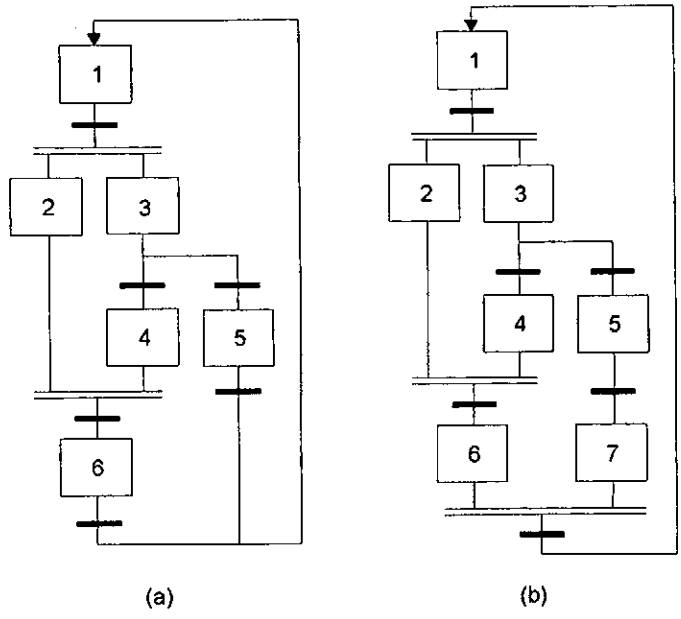


Figura 3.57. Exemplos de: (a) SFC inseguro; e (b) SFC con etapas inalcanzábeis.

3.6.2. Comentarios sobre o estándar IEC 61131-3

A pesar das numerosas avantaxes que os promotores do estándar preconizaron, como por exemplo:

- *A eliminación da dependencia dun único fabricante*, ao favorecer o estándar a uniformización das características ofrecidas polos diferentes fabricantes.
- *Un mellor aproveitamento dos recursos humanos*, xa que cun menor esforzo en formación conséguese unha maior aplicabilidade dos coñecementos adquiridos.
- *A reutilización dos programas de control e a dispoñibilidade de librarías xenéricas*, o estándar favorece a aparición de terceiras compañías que proporcionen solucións "software" independentes dos fabricantes de equipamentos "hardware".
- *Facilítase a programación e a depuración*, o estándar promove a programación estructurada, a utilización de tipos e estructuras de datos, a combinación de diferentes linguaxes, a utilización de librarías, etc. o que permite reducir os tempos de desenvolvemento, a presencia de erros e os custes de instalación e mantemento.

- *Posibilítase a adopción de solucións de control heteroxéneas*, ao favorecer a aparición de terceiras compañías que proporcionen equipos que poidan integrarse facilmente.
- *O estándar proporciona unha interface de programación uniforme* que facilita a evolución dos sistemas de control baseados en PLCs a sistemas máis flexíbeis e heteroxéneos, nos que se combinen diferentes equipamentos de control: p.e. PLC-PC, "SoftPLC", control baseado en PC, etc.

A día de hoxe o proceso de adopción do estándar está a ser lento e desigual, e pode dicirse que non se acadaron os niveis de aceptación que serian previsíbeis se se consideran os beneficios anunciados. As principais reticencias proveñen das comunidades de control americana e asiática nas que teñen maior importancia a presión do mercado e a esixencia de competitividade que o cumprimento do estándar, que é percibido como unha cuestión secundaria. A opinión maioritaria é a de estar a expectativa e agardar a que haxa mais productos dispoñíbeis no mercado de cara a considerar a súa adopción no futuro [14][167]. Entre as diferentes críticas que se teñen realizado [42][138] poden indicarse, por exemplo:

- *A estandarización dun conxunto de linguaxes constitúe un impedimento* á innovación e en último termo vai en contra dos intereses dos usuarios.
- *As linguaxes estandarizadas* (en concreto IL, FB, LD) *están anticuadas* e son linguaxes de baixo nivel baseadas en conceptos procedentes principalmente do campo da electromecánica e a electrónica.
- *A portabilidade completa dos programas é difícil de acadar*, e probablemente non se chegue nunca a conseguir completamente.
- *A adopción do estándar só proporciona beneficios a longo prazo*, xa que a substitución dun sistema de control propietario por outro baseado no estándar non proporciona ningún beneficio inmediato.

No que ao SFC respecta, valórase a súa capacidade de síntese de secuencias complexas mediante unha representación gráfica de fácil comprensión, mesmo para os non iniciados. As principais críticas que se lle realizaron foron as seguintes [42]:

- A utilización conxunta do SFC e as outras linguaxes do estándar obriga ao programador a traballar simultaneamente en dous planos: o secuencial definido polo SFC, e o combinacional, especificado nas accións utilizando as outras linguaxes.
- O conxunto de tipos de accións definido polo estándar é innecesariamente arbitrario e complexo, o que complica a interpretación temporal do seu comportamento e constitúe unha probable fonte de erros difíciles de detectar. Esta complexidade non está xustificada se se compara coa funcionalidade que proporcionan.

Sen embargo e a pesar destas críticas a percepción xeral é a de que o SFC presenta máis avantaxes que inconvenientes, por exemplo:

- Permite representar secuencias de control complexas mediante un conxunto reducido de estructuras básicas: secuencia, selección de secuencia, paralelismo e saltos de secuencia.
- Combinado co modelo software proposto no estándar promove a estructuración e a reutilización dos programas de control.
- Proporciona unha forma de especificar os contidos de receptividades e accións mediante as outras linguaxes definidas no estándar.
- Clarifica os tipos de accións e a semántica da súa execución.

3.7. Conclusións

Este capítulo dedicouse a presentar o Grafcet, que é o formalismo gráfico utilizado para a especificación de secuencias de control complexas na ferramenta proposta nesta tese de doutoramento. Expuxéronse as vantaxes que proporciona e as principais liñas de investigación relacionadas coa súa utilización como parte de metodoloxías de desenvolvemento "software", a proposta de extensións ao modelo, a verificación e validación mediante técnicas formais, etc. Describiuse a súa sintaxe, discutíronse os aspectos temporais das diferentes posibilidades de interpretación dos modelos Grafcet, e incluíronse unha formalización da relación entre o modelo e o seu contorno, unha descrición alxébrica da sintaxe e regras de evolución, e unha comparativa con outros formalismos de especificación de sistemas secuenciais. Mostráronse algúns exemplos da aplicación do Grafcet no modelado de sistemas industriais reais, e finalmente, explicouse a relación entre o SFC proposto no estándar IEC 61131-3 e o Grafcet.

O SFC é unha versión do Grafcet adaptada á programación de PLCs que, en resume, presenta as seguintes diferencias:

- O SFC elimina as posíbeis situacións de paralelismo interpretado (§3.2.3.10) nas seleccións de secuencia mediante a utilización de prioridades implícitas ou explícitas.
- O SFC restrinxe a unha o número de etapas iniciais en cada rede SFC —que é o equivalente a un grafcet conexo (§3.2.2.3)—.
- Os elementos sintácticos do SFC son unicamente os que permiten representar as estructuras de control básicas (§3.2.1) definidas no estándar IEC 60848, sen incluír ningunha das extensións sintácticas (§3.2.2) propostas posteriormente.
- A estructuración dos modelos SFC realízase mediante a división en POUs do modelo software proposto no estándar IEC 61131-3, no que as comunicacións entre SFCs realízanse mediante o intercambio de variábeis globais. Non hai equivalente no SFC ás xerarquías estructurais (§3.2.2.3) e de forzado (§3.2.2.4) do Grafcet.
- A semántica adoptada polo SFC é a SRS (con pequenas variacións), e presenta os problemas comentados na sección (§3.3.2).

Capítulo 4. Análise de aplicacións Grafcet

Dende a proposta inicial do Grafcet [1] foron varias as iniciativas de implementación de ferramentas para automatización de procesos que o incluíran como formalismo de especificación de secuencias de control. En [16] analízase, dende un punto de vista histórico, a incidencia do Grafcet no mercado norteamericano das ferramentas de automatización de procesos. Algunhas das aplicacións comentadas nese artigo son dos finais dos anos 80 e xa non están dispoñíbeis na actualidade (p.e. Sylgraf, Flexis) mentres que outras (p.e. Cadepa[27], PCPVirgo) foron adaptándose ás innovacións do mercado e na actualidade están dispoñíbeis comercialmente como ambientes para o desenvolvemento de aplicacións de control.

O obxectivo deste capítulo é analizar a situación actual no que ao soporte do Grafcet se refire. Para elo escolleuse unha mostra de aplicacións pertencentes a diferentes ámbitos que o utilizan para a especificación ou programación de secuencias de control. A escolla das aplicacións fíxose seguindo tres criterios:

1. *A súa representatividade* dentro dun ámbito de aplicación.
2. *A súa difusión,* xa sexa comercial ou educativa.
3. *A súa relevancia* como elemento de comparación para os obxectivos desta tese de doutoramento.

Debe terse en conta que esta escolla está limitada por dous factores inevitábeis: o coñecemento parcial do autor, dado o grande número de aplicacións existentes no mercado, e a dispoñibilidade dunha versión da aplicación que puidese ser analizada. As aplicacións escollidas, clasificadas por criterio de ámbito de aplicación[28], son as seguintes:

- *Programación de PLCs*: ActWin e PL7.
- *Desenvolvemento de aplicacións de control*[29]: AutomGen, Graf7-C e WinGrafcet.
- *Desenvolvemento de aplicacións de control distribuídas*: IsaGraph.
- *Desenvolvemento de aplicacións HMI/SCADA*: MachineShop e Visual I/O.
- *Librarías Grafcet*[30]: GrafcetView.

---

[27] Cadepa® é un producto da compañía Famic Technologies. http://www.famictech.com/

[28] Esta clasificación é unicamente orientativa, algunha das aplicacións poderían ser incluídas en varias categorías.

[29] Nesta categoría tentou incluírse a Cadepa, unha aplicación xurdida da investigación sobre Grafcet realizada na universidade francesa de Nancy nos finais dos 80, mais non se dispuxo dunha versión de demostración.

Un aspecto que é preciso ter en conta na análise é a proposta do SFC (§3.6) como linguaxe gráfica para a programación de secuencias de control en PLCs no estándar IEC 61131-3 e a súa influencia nas características do Grafcet implementadas nas ferramentas. Na actualidade a conformidade co estándar é percibida en xeral como un valor engadido nos productos, sobre todo a longo prazo, polo que a maioría das aplicacións tenden a implementar a versión aprobada nas recomendacións do estándar, en prexuízo da maior potencialidade de modelado do Grafcet. Estas aplicacións presentan un menor interese dende o punto de vista dos obxectivos desta tese de doutoramento.

O resto do capítulo está organizado da forma seguinte: no apartado (§4.1) explícase como se organizou a análise das aplicacións e os métodos utilizados nela; o apartado (§4.2) describe en detalle a análise de cada aplicación e, por último, no apartado (§4.3) resúmense os resultados e establécense as conclusións.

4.1. Estructuración e métodos utilizados na análise

Na análise das aplicacións seleccionadas tivéronse en conta tanto os aspectos sintácticos coma os semánticos do Grafcet, e relacionáronse as características implementadas coas recomendacións dos estándares IEC 60848 [84] e IEC 61131-3 [86]. Dividiuse a información de cada aplicación nos apartados seguintes:

- *Descrición e compoñentes*, no que se describe de forma xenérica a aplicación e as súas compoñentes e funcionalidades principais, dándolle maior importancia ao soporte ofrecido para a estructuración dos proxectos.
- *O editor Grafcet*, este é o apartado principal no que se analizan as características do editor gráfico e as capacidades de modelado, simulación e execución da aplicación. Pola súa extensión este apartado dividiuse nas seguintes seccións:
  1. *Estructuras de control*, dedicada a analizar as capacidades de edición das estructuras sintácticas básicas (§3.2.1), as extensións ao modelo incluídas (§3.2.2) e o uso das seleccións de secuencia, paralelismo, ciclos e saltos.
  2. *Estructura xerárquica*, na que se analiza o soporte ofrecido á edición tanto da xerarquía estructural (§3.2.2.3) como da de forzado (§3.2.2.4).
  3. *Accións e receptividades*, nesta sección comprobáronse diferentes aspectos relacionados coa edición de accións e receptividades como os tipos de accións incluídos, a sintaxe utilizada, o soporte de eventos, temporizadores e contadores, etc.
  4. *Identificación*, na que se analizan as opcións de identificación e as funcionalidades relacionadas coa renumeración automática durante a edición.
  5. *Análise sintáctica*, onde se analizou o método de análise sintáctica e comprobouse se se detectaban os erros mais comúns.
  6. *Simulación e execución*, sección na que se describen as opcións incluídas pola aplicación para a simulación e depuración dos programas.
  7. *Algoritmo de interpretación*, analízanse aquí a semántica da execución dos modelos Grafcet: tipo de interpretación (§3.3.2.2), resolución de conflictos entre accións (§3.3.3.8), aplicación de ordes de forzado (§3.3.2.5), etc.
- *Conclusións*, onde se resumen as principais características da aplicación.

[30] Non se incluiu o GraphChart [10] nesta categoría ao non disporse dunha versión de demostración.

No resto deste apartado descríbense en detalle as características analizadas detallando os métodos utilizados para realizar as comprobacións.

4.1.1. Características sintácticas

Coa intención de sistematizar a análise, dividíronse as características sintácticas do Grafcet en cinco grupos: estructuras de control básicas, as definidas no estándar IEC 60848; extensións das estructuras de control básicas, as aparecidas con posterioridade á norma IEC 60848; accións e receptividades, identificación e estructuras xerárquicas. Ademais analizáronse aspectos adicionais relacionados co funcionamento dos editores Grafcet que quedan fora do alcance do definido nos estándares:

- *As limitacións dos editores gráficos.* É bastante común nos editores Grafcet que o área de edición estea composta por unha matriz de celas organizadas en filas e columnas. O común é que neste tipo de editores sexa difícil ou mesmo non sexa posíbel editar estructuras complexas —que inclúan semáforos (§3.2.3.8), por exemplo—. Ademais algunhas aplicacións limitan o tipo de nodos que poden inserirse en cada cela dependendo da fila á que pertenza (p.e. filas impares para as etapas e pares para as transicións).
- *O método de verificación sintáctica.* Ademais de comprobar que os modelos cumpren a regra sintáctica da alternancia etapa-transición (unha etapa non pode estar unida directamente a outra etapa, nin unha transición a outra transición) os editores Grafcet teñen que verificar outros aspectos, como o código de accións e receptividades, a identificación de etapas e transicións, a coherencia lóxica das xerarquías estructural e de forzado, etc. En xeral hai dous métodos para realizar esta verificación: "on-line" (simultaneamente coa edición do modelo) ou "off-line" (unha vez que a edición do modelo está rematada). O mais habitual son as aproximacións mixtas, que realizan automaticamente a verificación "on-line" dalgunhas características (a regra de alternancia etapa-transición, a identificación de etapas e transicións) e o resto en "off-line" cando o usuario o solicita.

4.1.1.1. Estructuras de control básicas

Neste apartado comprobouse a utilización dos elementos sintácticos básicos definidos no estándar IEC 60848: etapas (incluídas as iniciais), transicións e unións mediante arcos orientados; así como as estructuras de control que poden modelarse con elas: secuencias, saltos, ciclos, seleccións de secuencia e paralelismo. As principais diferencias danse entre as aplicacións que implementan as recomendacións do estándar IEC 61131-3 e as que non. Así entre as primeiras hai aplicacións que permiten a edición dunha única rede SFC ou de varias, así como aplicacións que limitan a unha o número de etapas iniciais utilizadas en cada rede SFC ou que non impoñen ningún limite. En canto ao segundo tipo de aplicacións, o habitual é que non teñan estas limitacións e que permitan editar varios grafcets conexos sen límite de etapas iniciais.

Outro aspecto analizado foi a utilización de saltos e ciclos. Comprobouse se a aplicación permitía a súa utilización e a forma utilizada para representalos (p.e. arcos orientados, referencias de salto). Tamén se comprobou se era posíbel iniciar múltiples saltos dende un mesmo punto dunha secuencia de control, se son permitidos os saltos a etapas de niveis xerárquicos diferentes e se a aplicación actualizaba correctamente as referencias de salto durante a edición dos modelos.

Por último, verificouse a utilización das estructuras de selección de secuencia e paralelismo, distinguindo dous casos:

1. *O caso básico (ou rixido)*, no que unicamente poden modelarse estructuras completas nas que cada comezo de selección (ou paralelismo) ten que ir acompañado dun fin de selección (ou paralelismo).
2. *O caso flexíbel*, no que os comezos e fins de selección (ou paralelismo) poden utilizarse sen restriccións.

As aplicacións do primeiro tipo non permiten modelar grafcets con estructuras complexas. Para comprobar esta característica utilizouse o modelo da Figura 3.53 que require, para poder ser editado, dunha utilización flexíbel das estructuras de selección de secuencia e paralelismo.

4.1.1.2. Extensións ás estructuras de control básicas

Dentro desta categoría inclúense as estructuras sintácticas definidas con posterioridade á norma IEC 60848: etapas (transicións) fonte e sumidoiro (§3.2.2.1) así como calquera outra característica adicional non estándar implementada. Comprobouse se o programa detectaba como errónea a definición dunha transición como fonte e sumidoiro simultaneamente e se as transicións fonte estaban sempre validadas durante a execución.

4.1.1.3. Accións e receptividades

Na análise de accións e receptividades comprobáronse os seguintes aspectos, relacionándoos coas definicións dos estándares IEC 60848 e IEC 61131-3:

- Os tipos de variábeis permitidos e a nomenclatura utilizada para asignar a súa posición en memoria.
- Os tipos de accións (§3.2.1.5) implementados e a semántica temporal de cada tipo.
- As linguaxes permitidas para a edición de accións e receptividades. Neste punto hai variedade de solucións: aplicacións que utilizan algunha ou todas as linguaxes do estándar IEC 61131-3, que implementan algunha linguaxe propietaria ou que integran unha linguaxe de programación de alto nivel como Pascal ou C.
- O soporte proporcionado á utilización de eventos, temporizadores e contadores.
- O mecanismo utilizado para impedir os 'efectos colaterais' durante a avaliación de receptividades (§3.6.1.2).
- As características non estándar incluídas pola aplicación.

Por último tamén se analizou o mecanismo utilizado para resolver os conflictos entre accións, aínda que este aspecto foi incluído no apartado dedicado ao algoritmo de interpretación ao tratarse dunha característica relacionada coa execución dos modelos.

4.1.1.4. Identificación

Neste apartado analízase a forma de identificar as compoñentes dun modelo: etapas, transicións, accións, grafcets parciais, etc. Ademais de comprobar qué elementos poden identificarse e qué tipo de identificador pode utilizarse, tamén se analizaron os erros detectados durante a verificación sintáctica: identificadores duplicados, elementos sen identificar, referencias de salto a etapas inexistentes, etc.; e as opcións dispoñíbeis durante a edición: asignación automática de identificadores, renumeración de etapas e transicións ao copiar ou eliminar partes dun modelo, actualización automática de referencias de salto e variábeis de etapa nas renumeracións, etc.

4.1.1.5. Estructuras xerárquicas

Neste apartado comprobouse qué elementos das xerarquías estructural (§3.2.2.3) e de forzado (§3.2.2.4) implementa cada aplicación, e as facilidades ofrecidas para a súa edición. Verificáronse algúns aspectos relacionados coa corrección sintáctica e a coherencia lóxica destas estructuras, como: utilización única e aniñamento das macroetapas, corrección dos contidos das macroexpansións, sintaxe das ordes de forzado, presencia de ciclos na xerarquía de forzado, etc. No apartado dedicado ao algoritmo de interpretación tamén se analizou a aplicación das ordes de forzado durante a execución dos modelos.

4.1.2. Características semánticas

A análise das características semánticas de cada aplicación dividiuse en tres seccións: o algoritmo de interpretación (§3.3.2.2), a semántica temporal das accións (§3.3.3) e a aplicación das ordes de forzado (§3.3.2.5).

4.1.2.1. O algoritmo de interpretación

Ningunha das aplicacións analizadas permite escoller o tipo de interpretación semántica dun modelo Grafcet, todas implementan unha única aproximación: xa sexa a SRS ou a ARS. Para determinar a interpretación utilizada en cada aplicación utilizouse o grafcet da Figura 4.1. Nas aplicacións con algoritmo tipo SRS executarase a acción asociada á etapa 2, activando a saída S0 durante un tempo igual ao de franqueamento da transición 2, cuxa condición de transición é sempre certa. Polo contrario, nas aplicacións con algoritmo ARS a situación na que queda o grafcet ao franquear a transición 1 non é estábel, polo que evolucionará desactivando a etapa 2 e activando de novo a 1 sen activar a saída S0.

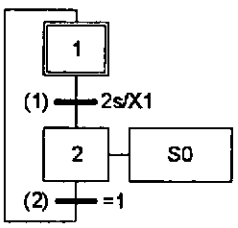


Figura 4.1. Grafcet utilizado para comprobar o tipo de interpretación.

Nas aplicacións con algoritmo ARS fixéronse dúas comprobacións adicionais:

1. Utilizouse o grafcet da Figura 4.2 para verificar como son considerados os valores de eventos e variábeis externas durante as evolucións internas (§3.3.2.6). Neste grafcet, partindo da situación inicial, o evento $\uparrow a$ pode producir catro evolucións diferentes[31]:
   a. $\{0, 3\} \rightarrow (1, 3) \rightarrow \{1, 4\}$, os eventos externos son considerados coma eventos durante a evolución interna e os internos son tratados no tempo interno.
   b. $\{0, 3\} \rightarrow \{1, 3\} \rightarrow \{1, 4\}$, os eventos externos son considerados coma eventos durante a evolución interna e os internos son tratados no tempo externo.
   c. $\{0, 3\} \rightarrow (1, 3) \rightarrow \{2, 4\}$, os eventos externos son considerados coma constantes durante a evolución interna e os internos son tratados no tempo interno.
   d. $\{0, 3\} \rightarrow (1, 3) \rightarrow \{2, 3\} \rightarrow (2, 4) \rightarrow \{2, 5\}$, os eventos externos son considerados coma constantes durante a evolución interna e os internos son tratados no tempo externo.

[31] As situacións estábeis representanse mediante chaves '{}'e as inestábeis mediante parénteses '()'.

2. Comprobouse se a aplicación incluía algún mecanismo para a detección de ciclos estacionarios durante as evolucións internas (§3.3.2.2). Para determinar esta característica utilizouse o grafcet da Figura 4.1, trocando a condición da transición 1 polo estado da variábel externa *a*, deste xeito mentres esa variábel estea activada, o grafcet entrará nun ciclo estacionario.

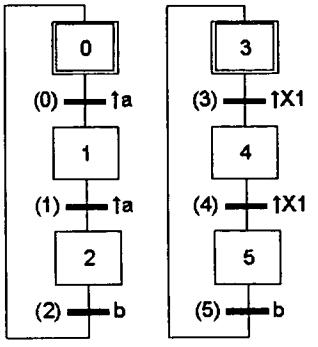


Figura 4.2 Grafcet utilizado para comprobar o manexo de eventos e variábeis durante as evolucións internas.

4.1.2.2. Semántica temporal das accións

Neste apartado analizouse o mecanismo utilizado en cada aplicación para resolver as situacións de conflicto entre accións (§3.3.3.8). Como se explica en (§3.6.1.3), o estándar IEC 61131-3 define un bloque de control de acción que determina o estado dunha variábel booleana ou acción cando hai simultaneamente varias etapas activas que a modifican. Ademais o estándar indica que a activación simultánea de varias temporizacións que afecten á mesma variábel booleana ou acción debe considerarse como errónea. Tendo en conta estas recomendacións analizáronse catro situacións diferentes:

1. Varias accións (tipos R, S e N) modifican simultaneamente a mesma saída booleana.
2. Varias accións temporizadas (tipos L, D, SL, SD, DS) modifican simultaneamente a mesma saída booleana.
3. Varias accións temporizadas (tipos L, D, SL, SD, DS) asignan valores diferentes á mesma variábel interna non booleana.
4. Varias accións simultáneas do mesmo tipo (non temporizadas) asignan valores diferentes á mesma variábel interna non booleana.

Dacordo ao estándar IEC 61131-3, no primeiro caso a acción tipo R ten prioridade sobre as demais; o segundo caso e o terceiro están prohibidos; e non se define o comportamento do sistema no último caso.

4.1.2.3. Aplicación das ordes de forzado

Neste apartado fixéronse, naquelas aplicacións que inclúen ordes de forzado, tres comprobacións adicionais que están relacionadas coa súa aplicación durante a interpretación dos modelos (§3.3.2.5):

1. As ordes de forzado son ordes internas que teñen que aplicarse tanto en situacións estábeis coma inestábeis. Para verificalo utilizouse o modelo da Figura 4.3. Nese grafcet, partindo da situación {1, 4, 8}, se é aplicada correctamente a orde de forzado da etapa 5 ao producirse o evento externo $\uparrow a$, o modelo debería evoluir do xeito seguinte: {1, 4, 8}→(2, 4, 8)→(2, 5, 9)→{2, 6, 7}.

2. A aplicación das ordes de forzado debe manter a coherencia lóxica da xerarquía de forzado, é dicir, as ordes son aplicadas recursivamente comezando polo nivel mais alto. Para comprobalo utilizouse o grafcet da Figura 3.37, no que partindo da situación {1, 10, 20, 30}, se as ordes de forzado das etapas 2 e 21 son aplicadas correctamente, ao activarse a variábel *a* producirase a evolución seguinte: {1, 10, 20, 30}→{2, 10, 21, 32}. E dende esta situación, ao producirse o evento $\uparrow b$, a evolución correcta sería unha das indicadas na Figura 3.37 (dependendo da interpretación dos eventos durante as evolucións internas).
3. Non están permitidas as situacións de forzado múltiple.

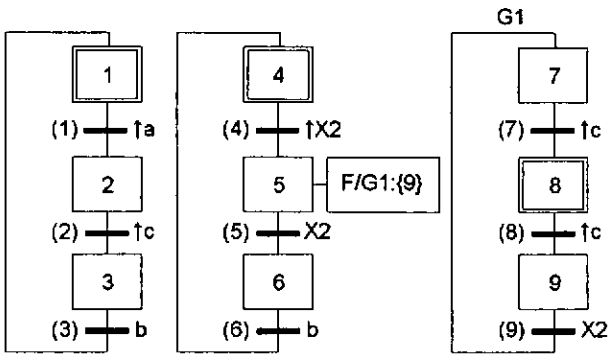


Figura 4.3. Grafcet utilizado para comprobar a aplicación das ordes de forzado durante as evolucións internas.

4.2. Aplicacións analizadas

4.2.1. GrafcetView

O GrafcetView é unha libraría desenvolvida pola empresa TecAtlant que permite a utilización de modelos Grafcet no LabView [56], un ambiente de programación da empresa National Instruments orientado ao desenvolvemento de aplicacións para a adquisición, procesamento e visualización de datos e o control de instrumentos. O proceso de desenvolvemento de aplicacións no LabView consiste na programación de instrumentos virtuais (VIs) mediante unha linguaxe gráfica propietaria denominada G. O LabView inclúe un grande número de VIs predefinidos para a adquisición de datos, o procesamento de sinais, o control de instrumentos, as comunicacións, operacións matemáticas, etc.

O GrafcetView implementa os VIs precisos para poder integrar modelos Grafcet nas aplicacións LabView, aproveitando deste xeito todas as facilidades de edición, depuración, visualización gráfica, conexión con dispositivos de E/S, etc. que proporciona esta aplicación. Mediante a implementación de librarías adicionais e os dispositivos externos axeitados sería posíbel implementar mediante LabView a funcionalidade dun PLC e programalo utilizando unha aproximación gráfica multilinguaxe semellante á proposta polo IEC 61131-3. A versión do GrafcetView analizada aquí é a 1.1 de demostración[32].

4.2.1.1. Compoñentes

A libraría GrafcetView implementa en dous VIs, un para as versións monotarefa e outro para as multitarefa, a estructura común (Figura 4.4) a todas as aplicacións que utilicen o Grafcet. Ademais, as operacións utilizadas para realizar a interpretación de modelos Grafcet están implementadas en cinco VIs adicionais:

[32] No resto deste apartado presuponse do lector unha familiaridade co manexo do LabView e a programación en G.

1. *G7-init*, iniciación da aplicación;
2. *G7-lecture*, lectura das entradas e cálculo de eventos e temporizacións;
3. *G7-calcul*, evolución do modelo Grafcet e cálculo das saídas;
4. *G7-delete*, finalización da aplicación;
5. *G7-calcul monoboucle*, implementa a funcionalidade conxunta de *G7-lecture* e *G7-calcul* nas aplicacións monotarefa.

O proceso de desenvolvemento dunha aplicación con GrafcetView realizase en tres fases:

1. *Definición das E/S da aplicación*. As E/S poden simularse mediante os controis e indicadores dos paneis de usuario ou adquirirse mediante algún dos dispositivos de E/S soportados polo LabView. Tantos as entradas como as saídas están definidas como arranxos de valores booleanos con identificadores coa sintaxe E_n para as entradas, e S_n para as saídas.
2. *Edición do grafcet de usuario* utilizando os VIs incluídos na libraría que implementan as estructuras de control do Grafcet.
3. *Integración dos resultados das dúas fases precedentes* nunha das aplicacións predefinidas polo GrafcetView. Para elo hai que inserir as entradas, saídas e modelos definidos polo usuario nos lugares indicados na estructura da Figura 4.4.

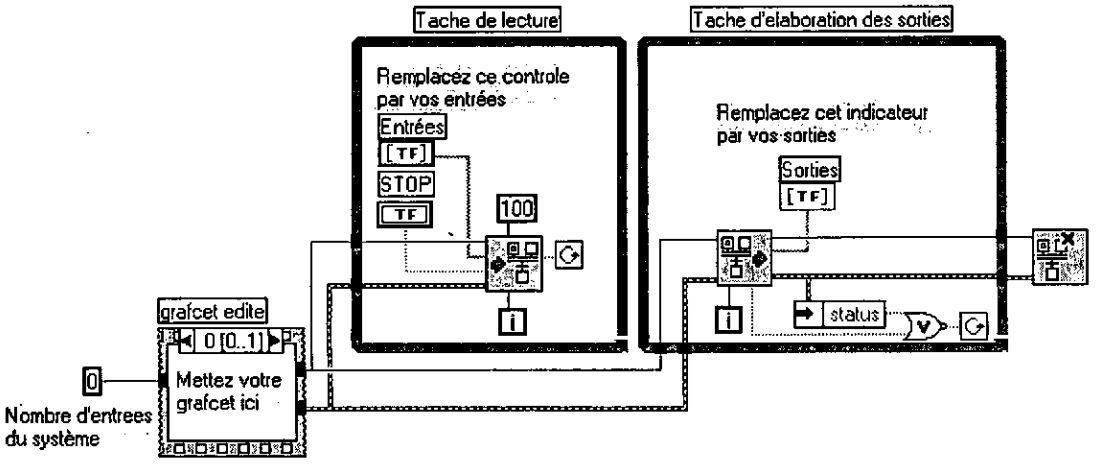


Figura 4.4. Contidos do VI que proporciona a estructura dunha aplicación GrafcetView (versión multitarefa).

4.2.1.2. O editor Grafcet

Ao ser unha libraría para LabView, o GrafcetView non dispón dun editor Grafcet propio, senón que proporciona un conxunto de VIs, controis e tipos de datos que permiten editar modelos Grafcet utilizando as facilidades proporcionadas polo ambiente gráfico do LabView.

4.2.1.2.1. Estructuras de control

GrafcetView inclúe controis para todos os elementos sintácticos básicos do Grafcet, coa excepción das estructuras de comezo e fin de selección de secuencia, que son realizadas mediante a conexión dunha etapa a múltiples transicións ou de múltiples transicións a unha etapa, respectivamente. Estes controis permiten modelar todas as estructuras de control básicas do Grafcet e, ademais, as estructuras de comezo e fin de paralelismo poden utilizarse de forma flexíbel, o que permite editar estructuras complexas.

A libraría tamén soporta algunha das extensións sintácticas como as etapas (e transicións) fonte/sumidoiro ou as macroetapas. Unha etapa pode utilizarse conectándoa só a transicións anteriores na secuencia de control (etapa sumidoiro), só a transicións posteriores (etapa fonte) ou sen conectala a ningunha transición. O mesmo acontece coas transicións, coa excepción de que unha transición si ten que estar conectada a, cando menos, unha etapa. De non ser así o GrafcetView o considera un erro de sintaxe. Durante as probas comprobouse que as transicións fontes non eran interpretadas correctamente ao non ser consideradas como validadas en todo momento.

Existen algunhas limitacións que afectan á edición dos Grafcets aínda que son de pouca importancia: os controis de comezo e fin de paralelismo non poden conectarse a mais de catro secuencias de control diferentes (é preciso aniñar varios controis cando se precisan mais); e os arcos orientados que van en sentido contrario á secuencia de control (de abaixo a enriba) teñen que conectar sempre unha transición a unha etapa.

4.2.1.2.2. Estructura xerárquica

A utilización de macroetapas basease nas funcionalidades do LabView para crear VIs mediante o capsulamento dos programas definidos polo usuario. Nos VIs creados deste xeito pode editarse tanto a icona que os representa graficamente como as conexións que proporcionan aos VIs de nivel superior. Utilizando este mecanismo poden crearse subgrafcets que agrupen calquera estructura que se utilice de forma habitual e definir as conexións utilizando catro tipos de datos incluídos na libraría. Estes tipos permiten identificar as conexións entrantes e saíntes e detectar automaticamente as que non cumpran coa regra de alternancia etapa-transición.

Se ben este mecanismo ten a avantaxe de ser moi flexíbel, é responsabilidade do usuario que as estructuras agrupadas cumpran coas regras definidas no estándar IEC 60848 para as macroetapas. Ademais non se inclúen as mesmas opcións de identificación para as macroetapas que para as etapas.

No que respecta ás particións do Grafcet, é posíbel utilizar o método anterior para capsular múltiples grafcets conexos e formar un grafcet parcial. Do mesmo xeito que acontecía coas macroetapas, isto non deixa de ser unha utilización específica dunha funcionalidade proporcionada polo LabView, e calquera característica adicional (como a identificación dos grafcets parciais) ten que ser implementada polo usuario. Ademais a libraría non inclúe ordes de forzado, polo que non é posíbel modelar unha xerarquía de forzado.

4.2.1.2.3. Accións e receptividades

No GrafcetView cada etapa pode ter asociada unha acción na que as saídas se indican mediante a notación S_n, separándoas con vírgulas no caso de querer activar varias simultaneamente. As accións poden levar asociada unha condición na que, mediante os operadores do álxebra de Boole: AND (.), OR (+) e NOT (-), poden editarse expresións booleanas que conteñan entradas (E_n), variábeis de etapa (X_n) e temporizacións sobre variábeis de etapa ($t_1/X_n/t_2$). O seguinte é un exemplo de acción condicional na que se activan dúas saídas:

S3, S4 si (E4+20/X2/50).-X3

As receptividades teñen unha sintaxe que consta de dúas partes: un evento e unha condición, separados pola palabra reservada '*et*'. Ambas partes son opcionais, e a ausencia simultánea dambas equivale a unha receptividade sempre certa. A condición segue a mesma sintaxe que a

explicada para as accións condicionais; encanto ao evento, este pode ser un flanco ascendente (*M*) ou descendente (*D*) aplicado a unha entrada, variábel de etapa ou temporización sobre unha variábel de etapa. Algúns exemplos de receptividades son os seguintes:

ME3 et (X2+E0)
D30/X4/60 et 10/X2/50
DX5 et –E5.-X12

Como pode deducirse da sintaxe de accións e receptividades, é posíbel mediante GrafcetView editar accións continuas, condicionais, limitadas e demoradas no tempo. Non se inclúen as accións impulsionais nin as memorizadas. Ademais a sintaxe das receptividades garante que non poidan producirse 'efectos colaterais' durante a súa avaliación.

4.2.1.2.4. Identificación

Co GrafcetView só é posíbel asociar identificadores ás etapas e estes deben ser numéricos. Non se inclúe ningunha opción de renumeración nin se actualiza esta automaticamente durante a edición do modelo para evitar duplicados. Tampouco se inclúe soporte para a numeración de macroetapas ou a identificación de Grafcets parciais.

4.2.1.2.5. Análise sintáctica

O GrafcetView realiza a análise sintáctica dos modelos en dúas fases. Durante a edición identifícanse as conexións que non cumpren a regra de alternancia etapa-transición, as etapas sen numerar ou os arcos orientados sen conectar; e unha vez que comeza a interpretación do modelo detéctanse os erros de sintaxe nas accións e receptividades, os identificadores de etapa duplicados ou as transicións non conectadas.

Un aspecto orixinal do GrafcetView consiste en que cada etapa é responsábel da análise sintáctica da acción que ten asociada (e cada transición da súa receptividade). Isto é posíbel porque tanto as etapas como as transicións están implementadas mediante controis (un tipo de VI) e, polo tanto, son realmente programas G representados no modelo mediante unha icona.

4.2.1.2.6. Simulación e execución

Os modelos Grafcet desenvolvidos co GrafcetView poden simularse e executarse do mesmo xeito que calquera outro programa en LabView. Sen embargo a división entre implementación (diagrama de bloques) e interface de usuario (panel) dos programas LabView non permite a animación da evolución dos modelos Grafcet directamente sobre a súa estructura. Existen dúas alternativas (Figura 4.5) para conseguir a animación do marcado das etapas activas:

1. Debuxar a estructura do grafcet nun panel de usuario e situar sobre o gráfico os indicadores de estado das etapas.
2. Utilizar dous indicadores que a libraría inclúe para mostrar os identificadores e o estado das etapas nos paneis de usuario durante a execución dos modelos.

O primeiro tipo de interface require que o usuario debuxe a estructura do grafcet e sitúe sobre ela os indicadores de etapa. Isto implica que un cambio no modelo require volver a debuxar a estructura e volver a situar os indicadores. En canto á segunda interface, é fácil de editar e modificar pero non sigue as recomendacións do estándar IEC 60848 e, no caso de grafcets complexos (con moitas etapas ou múltiples grafcets conexos), resulta complicado interpretala. Ningún dos dous tipos de solucións é completamente satisfactoria, mais en

contrapartida, o LabView facilita a edición de interfaces de usuario nas que se mostre o estado do proceso e que inclúan controis e indicadores que permitan interactuar con el.

4.2.1.2.7. O algoritmo de interpretación

GrafcetView utiliza un algoritmo de interpretación de tipo ARS, no que as accións son executadas unicamente nas situacións estábeis, e detecta e avisa da existencia de ciclos estacionarios. Os eventos externos foron considerados como eventos na escala interna, e os eventos internos foron tratados internamente (§4.1.2.1, caso 1.a).

En canto ás probas para comprobar a resposta diante de conflictos entre accións, no GrafcetView unicamente ten sentido realizar a segunda (§4.1.2.2). O resultado obtido foi que a aplicación non considerou coma un erro a existencia de múltiples temporizacións activas sobre a mesma variábel, e na saída obtívose un valor equivalente á aplicación da disxunción lóxica das accións activas.

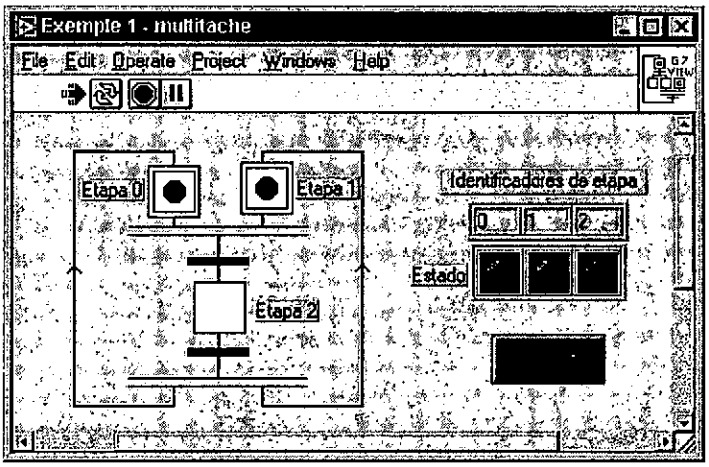


Figura 4.5. Dúas posíbeis interfaces para a animación das evolucións dun grafcet no GrafcetView.

4.2.1.3. Conclusións

GrafcetView é unha libraría que permite a utilización de modelos Grafcet como parte das aplicacións LabView. Inclúe soporte a todas as estructuras de control básicas, permite o uso flexíbel das seleccións de secuencia e paralelismo e a utilización de etapas (transicións) fonte/sumidoiro. A estructuración xerárquica basease nas capacidades do LabView para o capsulamento dos programas, non se inclúen ordes de forzado e queda baixo responsabilidade do usuario a utilización correcta das macroetapas e Grafcets parciais. Mediante unha sintaxe simple poden editarse accións continuas, condicionais, limitadas e demoradas no tempo. Nas condicións e receptividades poden utilizarse tanto temporizacións coma eventos. A análise sintáctica realizase en dúas fases e a interpretación dos modelos é de tipo ARS con detección de ciclos estacionarios, considerando os eventos externos como eventos na escala interna e resolvendo os conflictos entre accións mediante a disxunción lóxica das accións activas.

En conclusión, GrafcetView permite engadir o control secuencial de forma simple nas aplicacións LabView. É tamén unha excelente ferramenta para a aprendizaxe e o prototipado rápido de aplicacións, xa que LabView facilita enormemente o desenvolvemento de interfaces de usuario, a adquisición de datos, a interacción con equipos de proceso e o intercambio de información con outras aplicacións. Entre os puntos febles pódense citar: o soporte limitado á estructuración xerárquica dos modelos, a imposibilidade de animar as evolucións dos Grafcets

sobre a súa estructura, os problemas no uso de flancos coas variábeis de etapa e a aplicabilidade limitada da libraría ao depender do ambiente LabView.

4.2.2. MachineShop

MachineShop é o nome co que a empresa CTC Parker Automation comercializa un conxunto de aplicacións para o desenvolvemento de sistemas SCADA en arquitecturas PC con sistema operativo Windows. O aspecto gráfico do MachineShop é o dunha barra de tarefas no escritorio de Windows (Figura 4.6) dende a que poden xestionarse os proxectos de automatización, a comunicación cos sistemas remotos e a creación e modificación dos programas de control e interfaces de operador.

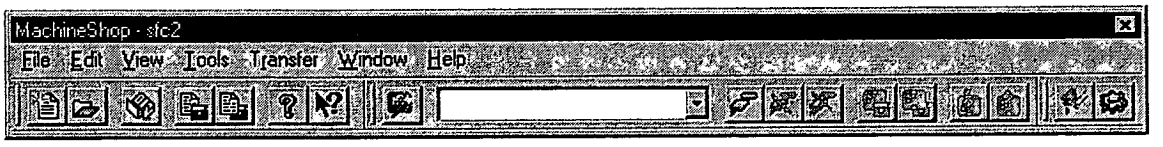


Figura 4.6. Interface da barra de tarefas do MachineShop.

4.2.2.1. Compoñentes

O programa para a definición da interface de operador denomínase Interact, e está formado por numerosos "drivers" de comunicación e módulos que proporcionan as funcionalidades precisas para implementar un sistema SCADA. Entre os módulos mais relevantes poden citarse:

1. *O editor de paneis de operador,* que permite editar interfaces que conteñan elementos gráficos como botóns, gráficas, indicadores, etc.
2. *O editor de animacións gráficas*, que permite incluír animacións gráficas nos paneis de operador.
3. *O xestor de alarmas, receitas, históricos e informes*, módulos que proporcionan paneis predefinidos para as opcións habituais nun SCADA en cada unha destas funcións.
4. *A configuración de máquinas*, que permite xestionar a carga e descarga de parámetros de configuración nos equipos de control.
5. *A transferencia de datos*, que permite configurar a transferencia de datos a alta velocidade entre diferentes equipos de control en aplicacións multicontrolador.
6. *As comunicacións*, módulo que permite compartir a información das aplicacións Interact con outras aplicacións a través de NetBios ou DDE.

Ademais destes módulos predefinidos, pode engadírselle á aplicación un programa de usuario que realice actividades adicionais. Este programa actívase mediante interrupcións e intercambia datos cos outros módulos a través dun área de memoria compartida.

A aplicación para o desenvolvemento do "software" de control denomínase MachineLogic (Figura 4.7) e sigue ás recomendacións do estándar IEC 61131-3. A versión de MachineLogic analizada neste apartado é a 2.01. Esta versión inclúe un ambiente de execución en tempo real no que poden simularse e depurarse os programas, e conta con "drivers" de E/S para buses de campo Profibus e DeviceNet. A interface do MachineLogic inclúe numerosas características que facilitan o seu manexo: barras de botóns, menús flotantes, xanelas que poden fixarse nos bordes do area de traballo, organización de múltiples xanelas mediante lapelas, asistentes á edición dos programas, etc. O elemento principal da aplicación é a árbore do proxecto dende a

que se ten acceso ás opcións principais de edición. Cada proxecto ten catro compoñentes principais:

1. *As POUs*, que poden ser de tres tipos: programas, funcións e bloques función dacordo ao definido no estándar IEC 61131-3. Cada POU divídese en tres partes: a descrición, a declaración de variábeis e a implementación, que pode codificarse utilizando calquera das linguaxes IEC: IL, ST, LD, FBD ou SFC.
2. *Os tipos de datos* do usuario: estructuras, arranxos e cadeas de caracteres definidos dacordo ao estándar IEC 61131-3.
3. *A configuración hardware*, que describe a estructura do sistema de control segundo o modelo "software" definido no estándar IEC 61131-3. O sistema pode conter varias *configuracións*, cada unha delas pode ter un ou mais *recursos*, e en cada recurso decláranse as variábeis globais, configúranse os dispositivos de E/S e execútanse unha ou mais *tarefas*. Cada tarefa executa unha ou mais POUs utilizando para cada unha un dos tres tipos de planificacións soportados: cíclica, de sistema ou de evento.
4. *As librarías*, que permiten a reutilización de POUs entre proxectos.

4.2.2.2. O editor Grafcet

O editor do MachineLogic está orientado á edición de SFCs que conteñan unha única rede cíclica na que non pode haber mais dunha etapa inicial. Utilízase a aproximación mais común neste tipo de editores consistente en dividir o área de traballo en filas e columnas e inclúense a maioría das opcións de edición habituais nun ambiente gráfico: seleccionar, mover, eliminar, desfacer os cambios, zoom, etc. O editor permite a inserción, eliminación e desprazamento dos nodos da rede unicamente se se mantén a súa corrección sintáctica e non se incumpre a restricción de ter unha única rede SFC.

4.2.2.2.1. Estructuras de control

Ao tratarse dun editor SFC, o MachineLogic ofrece soporte unicamente aos elementos sintácticos básicos do Grafcet, aínda que tamén inclúe a posibilidade de utilizar etapas sumidoiro para terminar secuencias de control non cíclicas. O editor non permite o cruce de liñas e a utilización de seleccións de secuencia e paralelismo só pode facerse mediante estructuras completas que comecen por un inicio de selección de secuencia (paralelismo) e rematen cun final de selección de secuencia (paralelismo). Esta restricción non permite modelar estructuras flexíbeis.

Está permitida a utilización de ciclos e saltos de secuencia, sendo posíbel iniciar múltiples ciclos ou saltos dende un mesmo punto da secuencia de control mediante a utilización dunha selección de secuencia. Ademais o editor permite definir unha etapa como *etapa de salto* a outra etapa. O identificador da etapa de salto ten que coincidir co da etapa á que se salta, aínda que o programa non controla durante a compilación que esta exista.

4.2.2.2.2. Estructura xerárquica

O editor non permite definir unha estructura xerárquica entre SFCs. A estructuración do proxecto realízase a nivel de POU, seguindo as recomendacións do estándar IEC 61131-3.

4.2.2.2.3. Accións e receptividades

O MachineLogic inclúe todos os tipos de accións IEC: N, R, S, P, L, D, DS, SD ou SL, e permite utilizar as linguaxes IL, ST, FBD e LD para implementar o seu contido. Os

identificadores das accións sen código son utilizados como variábeis booleanas e o seu valor é modificado durante a activación da etapa dacordo ao tipo de acción do que se trate.

As receptividades tamén poden implementarse utilizando as mesmas linguaxes que para as accións. No caso de utilizar LD ou FBD, a implementación da receptividade pode facerse directamente no editor SFC. Entre os FBs dispoñíbeis hai contadores, temporizadores, biestábeis e detectores de flancos, e tamén pode utilizarse o estado de activación das etapas (coa sintaxe $step_n.X$) no código de receptividades e accións. O programa non impide a edición de receptividades que produzan 'efectos colaterais' durante a súa avaliación.

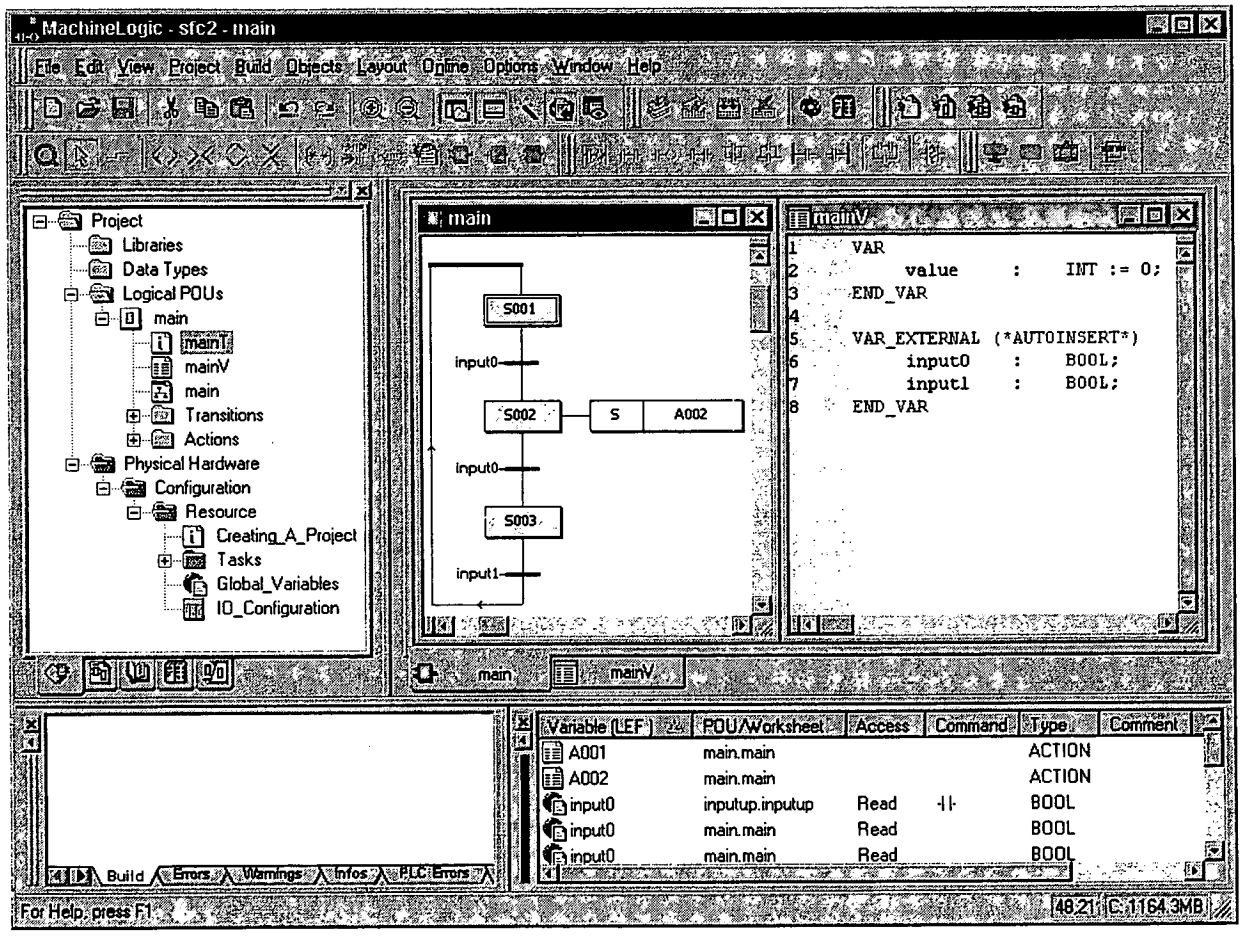


Figura 4.7. Interface do MachineLogic.

4.2.2.2.4. Identificación

O MachineLogic permite a asignación de identificadores alfanuméricos únicos a etapas, transicións e accións sen que existan conflictos entre eles (é posíbel ter unha etapa, unha transición e unha acción co mesmo identificador). O programa detecta os identificadores duplicados durante a compilación.

4.2.2.2.5. Análise sintáctica

A análise sintáctica faise en dúas fases, durante a edición dos SFCs o editor non permite o cruce de liñas nin operacións que incumpran a regra de alternancia etapa-transición. O resto de comprobacións realízanse durante a compilación do proxecto. O único problema atopado foi a non comprobación da existencia da etapa destino nas etapas de salto.

4.2.2.2.6. Simulación e execución

O MachineShop inclúe un subsistema implementado sobre unha extensión ao sistema operativo DOS, denominada RTXDOS, que proporciona capacidades multitarefa e de tempo real. Este subsistema está formado por un compilador, un xestor da execución das tarefas de control que emula o funcionamento dun PLC, un depurador, un xestor de erros e os "drivers" de E/S e comunicacións. Dende o MachineLogic pódense configurar, compilar, descargar, simular e depurar en tempo real as POUs do proxecto. Durante a execución disponse de numerosas opcións para a depuración e monitorización dos programas: inicio e parada de recursos, monitorización e forzado de variábeis, animación gráfica dos elementos das redes LD, FBD e SFC, modificación "on-line" das POUs (con certas restriccións), simulación de E/S, etc.

4.2.2.2.7. O algoritmo de interpretación

O algoritmo de interpretación utilizado no subsistema en tempo real do MachineLogic é de tipo SRS, executándose as accións tanto en situacións estábeis como inestábeis. Os resultados das probas realizadas para comprobar o funcionamento do programa cando hai conflictos entre accións (§4.1.2.2) foron os seguintes: no caso (1.a) a saída mantivo o valor 0, o que indica que a acción R tivo prioridade sobre as demais segundo o definido no estándar IEC 61131-3; no caso (1.b) o programa non considerou que a situación dada fose errónea e a saída tomou o valor resultado da disxunción lóxica das accións activas; e, por último, os casos (1.c) e (1.d) tampouco foron considerados coma erros e á variábel asignáronselle valores de forma errática mentres as accións estiveron activas.

4.2.2.3. Conclusións

O MachineShop é unha aplicación para o desenvolvemento de sistemas SCADA composta por tres compoñentes principais: un ambiente de execución en tempo real, un editor (o MachineLogic) para a implementación de programas de control, e un editor (o Interact) para a interface de usuario e demais aspectos do sistema (alarmas, receitas, históricos, etc.). O MachineLogic é un ambiente gráfico que sigue as recomendacións do estándar IEC 61131-3 e inclúe numerosas opcións para estructurar o "software" e configurar o "hardware", as comunicacións e os dispositivos de E/S dun proxecto. Para a programación das POUs pódense utilizar todos os tipos de accións e linguaxes IEC, e as opcións de simulación e depuración son moi completas: monitorización e forzado de variábeis, definición de puntos de parada ("breakpoints"), animación do estado dos SFCs, simulación das E/S, etc.

En canto ás características Grafcet que o MachineLogic implementa, estas limítanse ás incluídas na definición do SFC coa única extensión sintáctica das etapas sumidoiro. Non se permite a estructuración xerárquica dos SFCs, o algoritmo de interpretación é de tipo SRS e non se manexan correctamente os conflictos entre accións en todos os casos analizados. En resume, o MachineShop é unha aplicación moi completa que facilita o desenvolvemento de sistemas SCADA en arquitecturas PC con sistema operativo Windows. Un inconveniente da aplicación é que o subsistema de execución en tempo real non é portábel e funciona unicamente en equipos co sistema operativo DOS instalado. No que respecta ao Grafcet, o MachineLogic non aporta ningunha característica nova, limitándose a implementar o SFC dacordo ao estándar IEC 61131-3.

4.2.3. IsaGraph

O IsaGraph da compañía AlterSys[33] é unha das aplicacións de desenvolvemento de software de control mais populares no mercado da automatización industrial. Pioneira na aplicación do concepto de automatización aberta (consistente na aplicación dos principios do software aberto á automatización industrial), permite o desenvolvemento de aplicacións independentes do sistema final no que vaian executarse. O único requisito é a dispoñibilidade dunha versión do ambiente de execución do IsaGraph para ese sistema concreto. A versión da aplicación aquí analizada é a 4.10. Esta versión soporta múltiples ambientes de execución distribuídos en diferentes equipos interconectados mediante unha ou mais redes de comunicacións, así como a posibilidade de incorporar un servidor "web" no sistema de control.

4.2.3.1. Compoñentes

O IsaGraph é unha aplicación complexa que inclúe numerosas funcionalidades e opcións. A súa arquitectura de alto nivel divídese en tres compoñentes principais:

1. *O ambiente de desenvolvemento*, con versións para Windows 98, 2000 e NT, proporciona soporte á configuración do sistema de control e á codificación, simulación e execución do seu "software". A arquitectura dos proxectos sigue as recomendacións do estándar IEC 61131-3: cada configuración IEC correspóndense no IsaGraph cun nodo físico que executa unha ou mais máquinas virtuais. Cada máquina virtual executa un recurso (o equivalente a un PLC virtual) e a súa versión determina as redes de comunicacións, dispositivos de E/S, funcións C e bloques función dispoñíbeis. En cada recurso configúranse as POUs e datos que conteña, os parámetros de execución, as comunicacións e os dispositivos de E/S. A compoñente principal do ambiente de desenvolvemento é o *Xestor de Proxectos*, dende o que se accede ao resto de opcións da aplicación. Entre as mais importantes destacan:
   a. *O editor "hardware"*, no que se configura o "hardware" do sistema de control: os parámetros das configuracións, das redes de comunicación e das conexións, a asignación de recursos a cada configuración, as propiedades de execución de cada recurso, etc.
   b. *O editor de enlaces entre recursos*, no que se editan os recursos e a definición dos enlaces[34] para o intercambio de información entre eles. Para cada recurso móstranse as POUs, datos e parámetros mediante unha estructura de árbore dende a que pode accederse aos editores do código das POUs. IsaGraph soporta todas as linguaxes IEC máis os diagramas de fluxo.
   c. *O diccionario do proxecto*, no que se edita a información do proxecto, que está organizada en catro categorías: parámetros de funcións e bloques función, variábeis, tipos de datos e macros definidas polo usuario.
   d. *O editor da configuración de E/S*, no que se edita a asignación de variábeis aos canais de E/S. Este editor inclúe opcións para engadir e eliminar dispositivos de E/S, configurar os parámetros de cada canal, asignar e eliminar variábeis, etc.

---

[33] O IsaGraph foi desenvolvido inicialmente pola compañía CJ International. Debido ao seu éxito comercial foi absorbida pola AlterSys, distribuidora do Virgo Automation Studio e líder na comercialización de productos softPLC e softDCS. A versión actual do IsaGraph é o resultado da integración dambos productos, da adaptación ás recomendacións do estándar IEC 61131-3 e do abandono comercial da denominación Virgo.

[34] O concepto de enlace é semellante ao de acceso ("access path") definido no estándar IEC 61131-3. A principal diferencia é que os enlaces intercambian información entre recursos que poden estar ou non na mesma configuración. Os accesos son para intercambios entre recursos de diferentes configuracións.

2. *A máquina virtual* ou ambiente de execución, no que se executan as aplicacións compiladas no ambiente de desenvolvemento. Esta máquina está deseñada para ser facilmente portábel, existindo versións para moitos dos sistemas operativos mais coñecidos, así como para diferentes tipos de equipamentos de control. A portabilidade da máquina basease na definición de catro interfaces que abstraen os detalles propios de cada sistema:
   a. *A interface de comunicacións*, utilizada para a depuración e a comunicación da máquina virtual con outros sistemas como SCADAs, HMIs, PLCs, etc.
   b. *A interface de E/S*, para o intercambio de información co proceso mediante tarxetas de E/S, buses de campo, etc.
   c. *A interface de sistema*, que abstrae os servicios do sistema operativo para a xestión de temporizadores, memoria, etc.
   d. *A interface de aplicación*, que proporciona acceso a funcións C e bloques función que permiten aumentar as funcionalidades da máquina virtual.
3. *As ferramentas de soporte*, orientadas aos usuarios avanzados que queiran adaptar a máquina virtual a un novo sistema ou incorporar novas funcionalidades.

4.2.3.2. O editor Grafcet

O editor do IsaGraph (Figura 4.8) está orientado á edición de SFCs que conteñan múltiples redes nas que non se restrinxe o número de etapas iniciais en cada rede. Utilízase a aproximación mais común neste tipo de editores consistente en dividir o área de traballo en filas e columnas e inclúense as opcións de edición habituais nos ambientes gráficos: seleccionar, mover, copiar, eliminar, desfacer os cambios, zoom, etc.

4.2.3.2.1. Estructuras de control

Ao tratarse dun editor SFC só ofrece soporte aos elementos sintácticos básicos do Grafcet. A utilización das seleccións de secuencia e paralelismo é flexíbel mais non o suficiente como para modelar estructuras complexas que inclúan semáforos, debido a que o editor non permite a conexión directa entre un inicio de selección de secuencia e un final de paralelismo ou entre un inicio de paralelismo e un final de selección de secuencia.

Os ciclos e saltos de secuencia poden representarse graficamente mediante arcos orientados ou mediante *referencias de salto*. O programa controla que as conexións que se realicen desta maneira cumpran a regra de alternancia etapa-transición. Se se queren iniciar múltiples saltos a diferentes etapas dende un mesmo punto da secuencia de control pode utilizarse unha estructura de inicio de selección.

4.2.3.2.2. Estructura xerárquica

O IsaGraph aporta unha solución propia que non coincide con ningunha das recomendacións dos estándares IEC, poden establecerse relacións de subordinación entre as POUs codificadas con SFC dividíndoas en dúas categorías:

1. *Os SFCs principais*, activados polo sistema ao comezo da execución.
2. *Os SFCs dependentes*, cuxa execución é controlada polos SFCs dos que dependen.

A Táboa 4-I resume as cinco accións que controlan a execución dun SFC dependente. Con este mecanismo poden implementarse funcionalidades semellantes, ainda que máis limitadas e non equivalentes semanticamente, ás das ordes de forzado ou aos "enclosure steps" —definidos na última revisión do estándar Grafcet [85]—.

4.2.3.2.3. Accións e receptividades

O IsaGraph non inclúe todos os tipos de datos nin de accións definidos polo estándar IEC 61131-3. En concreto permite accións dos tipos: N, R, S, P0 e P1 (non inclúe accións temporizadas) e divídeas en dúas categorías:

1. As que poden programarse mediante unha das linguaxes IL, ST ou LD (tipos N, P0 ou P1).
2. As booleanas (tipos R, S e N), que poden utilizarse para activar e desactivar o valor de variábeis booleanas ou nomes de SFCs (equivalen ás accións GSTART e GKILL).

En canto ás receptividades, poden programarse mediante IL, ST ou LD e poden utilizarse as funcións booleanas e FBs dispoñíbeis, entre os que hai contadores, temporizadores, biestábeis e detectores de flancos. O IsaGraph non impide os 'efectos colaterais' que poidan producirse durante a avaliación das receptividades, e define dúas variábeis para cada etapa — o estado de activación ($step_n.X$) e o tempo de actividade ($step_n.T$)— cuxos valores poden ser utilizados no código de accións e receptividades.

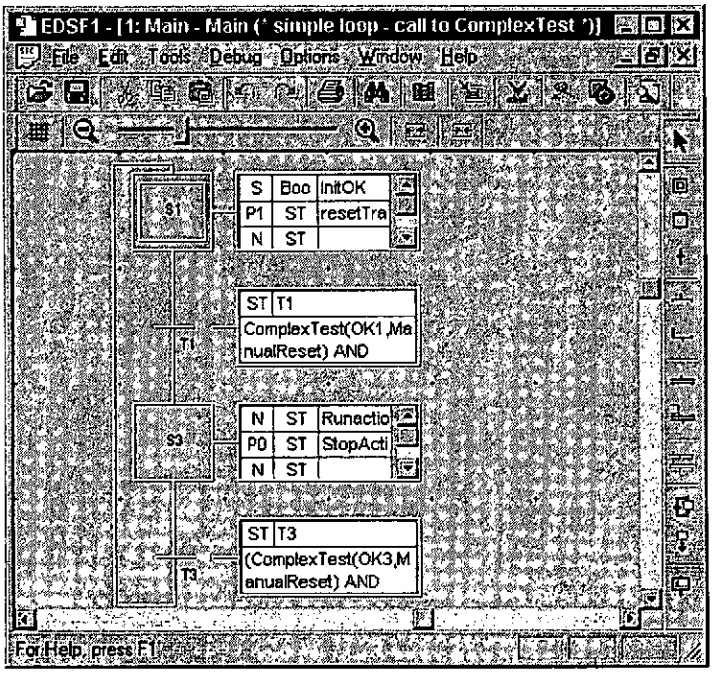


Figura 4.8. Interface do editor SFC no IsaGraph.

GSTART	Activa as etapas da situación inicial do SFC dependente.
GKILL	Desactiva todas as etapas activas no SFC dependente.
GFREEZE	Memoriza a situación actual do SFC dependente.
GRST	Activa as etapas da última situación memorizada.
GSTATUS	Consulta o estado do SFC dependente (activo, suspendido ou inactivo).

Táboa 4-I. Accións que controlan a execución dos SFCs dependentes no IsaGraph.

4.2.3.2.4. Identificación

O IsaGraph numera automaticamente as etapas e transicións dos SFCs asignándolles identificadores alfanuméricos únicos (S_n para as etapas e T_n para as transicións), e permite que sexan modificados polo usuario. A unicidade dos identificadores unicamente é obrigatoria en cada SFC, e a detección de identificadores duplicados realízase durante a compilación do

programa. Algunhas características relacionadas coa identificación que facilitan a edición de SFCs son:

1. A renumeración automática de etapas e transicións a petición do usuario.
2. A renumeración automática durante as operacións de copia para evitar duplicados.

Ambas opcións actualizan correctamente as referencias de salto e a primeira dêlas tamén actualiza correctamente as variábeis de etapa utilizadas no código de accións e receptividades.

4.2.3.2.5. Análise sintáctica

A análise sintáctica faise en dúas fases, durante a edición comprobase que as conexións cumpran coa regra da alternancia entre etapas e transicións e durante a compilación realízase a análise sintáctica completa.

4.2.3.2.6. Simulación e execución

O IsaGraph inclúe opcións para a compilación, simulación e depuración das aplicacións. Pode compilarse un proxecto completo, un recurso ou unicamente unha POU. A compilación pode xerar un código intermedio independente do sistema (TIC), ou ben código C que pode ser incluído nas librarías da máquina virtual. Durante a simulación os recursos son emulados mediante máquinas virtuais que se executan no mesmo computador que o ambiente de desenvolvemento. A interacción co proceso realizase a través de paneis de E/S dende os que se poden visualizar, modificar e forzar os valores das variábeis, entradas e saídas. Entre outras opcións dispoñíbeis, durante a simulación pode verse graficamente o estado de activación das etapas, indicar puntos de parada ("breakpoints"), forzar o franqueamento de transicións, etc.

Para a depuración da aplicación é preciso descargar previamente os recursos compilados nas máquinas virtuais definidas durante a configuración do sistema. Estas máquinas teñen a capacidade de almacenar o código dos recursos en disco, o que permite recuperalos posteriormente dende o ambiente de desenvolvemento ou iniciar a súa execución de forma autónoma (sen descarga previa). Entre as opcións que as máquinas virtuais proporcionan durante a depuración poden citarse: o inicio e parada de recursos, a execución continua ou ciclo a ciclo, a información de estado (tempos de ciclo, variábeis de estado) e a diagnose dos recursos.

4.2.3.2.7. O algoritmo de interpretación

A máquina virtual do IsaGraph executa os programas utilizando un algoritmo de interpretación de tipo SRS. Nas probas realizadas para comprobar a resposta do IsaGraph cando varias accións modifican simultaneamente unha mesma variábel (§4.1.2.2), os resultados foron os seguintes: no caso (1.a) a saída mantivo o valor 0, o que indica que a acción R tivo prioridade sobre as demais dacordo ao definido no estándar IEC 61131-3; os casos (1.b) e (1.c) non foron analizados pois o IsaGraph non ten accións temporizadas; e, por último, o caso (1.d) non foi considerado coma un erro, asignándoselle valores de forma non determinista á variábel mentres as accións estiveron activas.

4.2.3.3. Conclusións

O IsaGraph é unha das aplicacións que con mais éxito aplica os conceptos de automatización aberta e "SoftPLC". O núcleo da aplicación está formado por unha máquina virtual facilmente portábel, que emula a arquitectura lóxica dun PLC sobre calquera equipo que dispoña dun sistema operativo multitarefa. O desenvolvemento dos proxectos realízase dende

un ambiente gráfico que incorpora parte das recomendacións do estándar IEC 61131-3 no referente á estructura da aplicación e á programación das POUs. Dende este ambiente pódense editar múltiples parámetros relacionados coa configuración "hardware", arquitectura "software", comunicacións e interfaces de E/S; programar, compilar e depurar as POUs de cada recurso; descargar o código compilado nas máquinas virtuais e monitorizar a súa execución. Para a programación das POUs, pódense utilizar todas as linguaxes IEC e máis os diagramas de fluxo.

En canto ás características do Grafcet incluídas no IsaGraph, limítanse á implementación dunha versión do SFC: non se inclúe ningunha das extensións sintácticas do Grafcet, a estructura xerárquica non sigue as recomendacións dos estándares IEC, non se inclúen todos os tipos de accións IEC, o algoritmo de interpretación é de tipo SRS e non se manexan correctamente os conflictos entre accións. En resume, o IsaGraph é unha aplicación moi completa para o desenvolvemento de proxectos de automatización en sistemas distribuídos que inclúe unha potente máquina virtual que facilita a creación de "software" portábel. No que respecta ao Grafcet, o IsaGraph non aporta ningunha característica nova, limitándose a implementar unha versión do SFC se exceptuamos a posibilidade de definir unha estructura xerárquica entre SFCs que non se axusta ao definido nos estándares Grafcet.

4.2.4. PL7

O PL7 é unha aplicación da empresa Schneider Automation para a programación de PLCs da marca Telemecanique dende PCs co sistema operativo Windows. O ambiente gráfico do programa sigue as directrices do estándar IEC 61131-3 adaptadas ás características dos PLCs que soporta. A versión analizada é a PL7 Pro v3.1, que permite traballar cos PLCs das series TSX Micro e TSX Premium.

4.2.4.1. Compoñentes

A compoñente principal da aplicación é o *navegador*, no que se organiza o proxecto de automatización nunha estructura de tipo árbore. Dende este navegador tense acceso a todas as demais funcionalidades do programa: definición da estructura dos programas, definición das variábeis e tipos de datos, programación, compilación e depuración das POUs, configuración dos PLCs e da comunicación, documentación dos programas, etc. A aplicación tamén inclúe un editor cunha libraría de obxectos gráficos (bombas, depósitos, motores, etc.) para crear interfaces de operador simples.

A execución dos programas nos PLCs TSX das series Micro e Premium pode facerse tanto en modo monotarefa como multitarefa, dependendo das capacidades de cada equipo concreto. En calquera caso sempre existe unha tarefa principal, denominada MAST, que pode executarse cíclica ou periodicamente. Nas configuracións multitarefa poden engadirse ademais tarefas de alta prioridade para a xestión rápida de eventos externos ou a realización de operacións que requiran un período inferior ao da tarefa MAST. Todas as tarefas divídense en seccións e subprogramas (que poden ser chamados dende as seccións ou dende outros subprogramas). Cada sección pode programarse utilizando as linguaxes IEC: IL, LD, FBD ou ST. Ademais, nalgúns PLCs pode incluírse na tarefa MAST unha sección adicional programada con Grafcet. Esta sección divídese en tres partes: o tratamento previo, o tratamento secuencial e o tratamento posterior. Os tratamentos previo e posterior poden programarse con IL, ST ou LD e serven para realizar operacións de iniciación e finalización do proceso secuencial executado polo grafcet.

4.2.4.2. O editor Grafcet

O editor Grafcet do PL7 (Figura 4.9) está orientado á edición de múltiples grafcets conexos cíclicos. Utiliza a opción de dividir o área de traballo en filas e columnas (reservando as filas impares para as transicións e as pares para as etapas) e soporta as operacións básicas de edición nun ambiente gráfico: seleccionar, mover, copiar, eliminar, zoom, etc. O tamaño do área de traballo está limitado a oito páxinas de 154 celas cada unha, organizadas en catorce filas e once columnas. O editor tamén limita o número de etapas, transicións e macroetapas que poden utilizarse dependendo do PLC escollido.

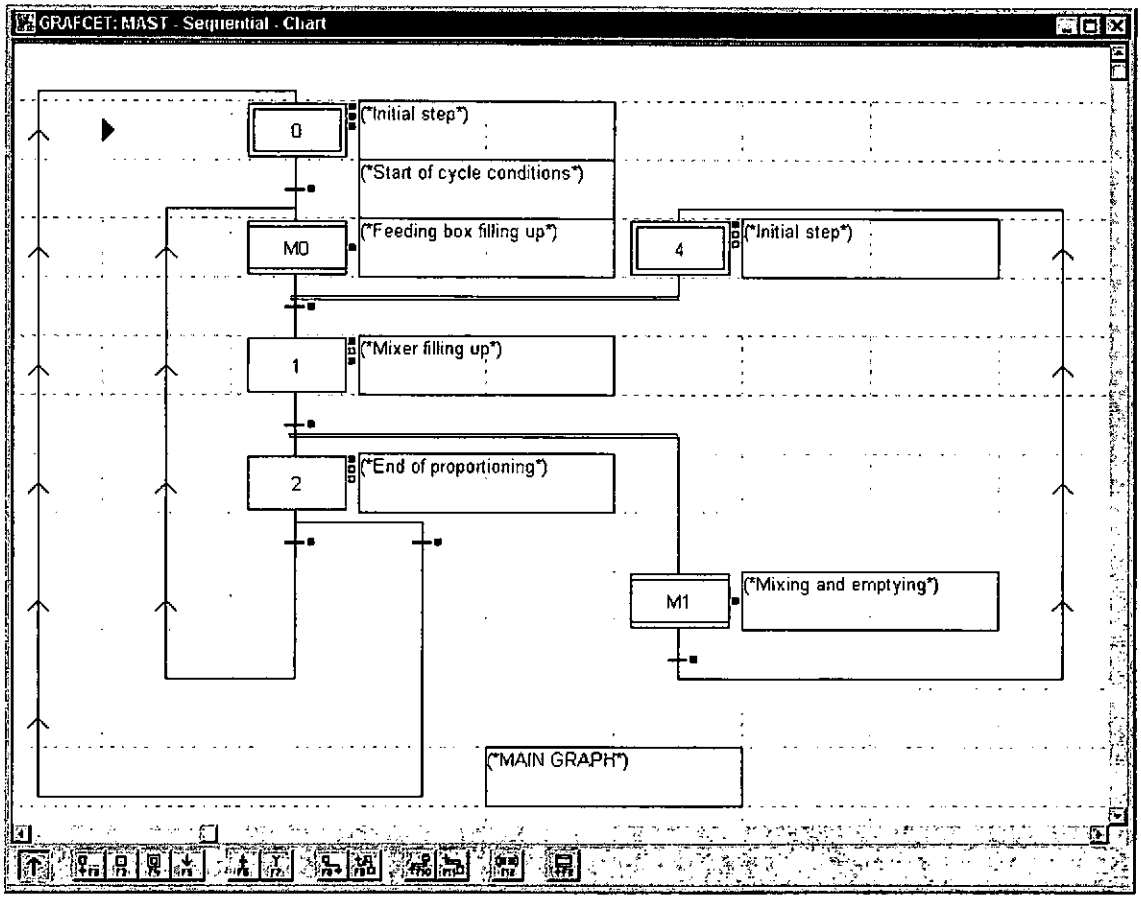


Figura 4.9. Editor Grafcet do PL7.

4.2.4.2.1. Estructuras de control

O editor Grafcet do PL7 soporta a edición de múltiples grafcets conexos cíclicos que conteñan calquera das estructuras de control básicas. O uso das seleccións de secuencia e paralelismo é flexíbel, aínda que a división do área de traballo en filas e columnas limita en certos casos a complexidade das estructuras que poden editarse. Os ciclos e saltos de secuencia poden representarse mediante arcos orientados ou mediante pares de frechas que indican a orixe e o destino do salto, estando permitido incluír múltiples saltos a diferentes etapas dende un mesmo punto da secuencia de control.

4.2.4.2.2. Estructura xerárquica

Dependendo do modelo concreto de PLC, o editor permite a utilización de macroetapas nos modelos e inclúe soporte á estructura xerárquica que estas forman. O límite de aniñamento das

macroetapas é de oito niveis, e a estructura xerárquica que forman pode verse no navegador da aplicación. O programa controla que cada macroetapa só sexa utilizada unha vez e que non existan recursividades nin ciclos na estructura xerárquica. As etapas de entrada e saída nas expansións das macroetapas denomínanse IN e OUT respectivamente, e só a etapa IN pode ter accións asociadas. As macroexpansións non están limitadas a unha única estructura conexa que comece na etapa de entrada e remate na de saída, senón que poden conter varios grafcets conexos, sempre e cando as etapas de entrada e saída sexan únicas.

4.2.4.2.3. Accións e receptividades

O PL7 permite a utilización nas accións e receptividades dos elementos de memoria dos PLCs Telemecanique: entradas, saídas, variábeis de sistema, variábeis de estado dos FBDs e Grafcets, etc. Tamén poden utilizarse os FBDs predefinidos que inclúen contadores, temporizadores e biestábeis. Unha parte da memoria de sistema resérvase para manter os parámetros de configuración do Grafcet e a información sobre o estado e tempo de activación de cada etapa durante a execución. O programa permite a modificación destas variábeis na sección de tratamento previo do grafcet mediante a utilización de instruccións SET e RESET.

As etapas poden ter asociadas tres accións, unha de activación (tipo P0), unha continua (tipo N1) e unha de desactivación (tipo P1), que poden ser programadas mediante as linguaxes IL, LD, FBD ou ST. A documentación do programa indica que as accións N1, aínda que se denominen continuas, son tratadas como accións memorizadas e será preciso activalas e desactivalas explicitamente mediante instruccións SET e RESET nas accións P0 e P1 para limitar o seu efecto á duración dunha etapa.

As receptividades tamén poden ser programadas coas mesmas linguaxes, aínda que o PL7 restrinxe as características que poden ser utilizadas para evitar os 'efectos colaterais' durante a súa avaliación.

4.2.4.2.4. Identificación

O PL7 só permite asignar identificadores numéricos ás etapas e non inclúe opcións de renumeración durante as operacións de copia. O programa detecta os duplicados durante a verificación sintáctica e permite, cando se utilizan macroetapas, que o mesmo identificador sexa utilizado en macroexpansións distintas.

4.2.4.2.5. Análise sintáctica

A análise sintáctica faise en dúas fases, durante a edición do modelo comprobase o cumprimento da regra de alternancia etapa-transición. A análise completa realízase durante a fase de verificación sintáctica unha vez rematada a edición.

4.2.4.2.6. Simulación e execución

O PL7 non inclúe a posibilidade de simular os programas, e a súa depuración e monitorización en tempo real require descargalos previamente no PLC. Inclúense opcións que durante a depuración permiten ver e forzar o valor dos elementos de memoria do PLC, ver graficamente a animación do estado dos programas (incluído o grafcet) e realizar accións relacionadas coa depuración da sección Grafcet, como a suspensión da evolución do modelo ou o forzado dunha situación específica.

4.2.4.2.7. O algoritmo de interpretación

O algoritmo de interpretación do PL7 é de tipo SRS, o propio do modo de funcionamento cíclico nos PLCs. As probas para comprobar a resposta do programa cando varias accións modifican simultaneamente unha mesma variábel (§4.1.2.2) non puideron facerse no PL7 ao non dispor o programa dun simulador.

4.2.4.3. Conclusións

O PL7 é unha aplicación para a programación de PLCs da marca Telemecanique que permite a utilización das linguaxes IL, LD, FBD, ST e Grafcet. O programa incorpora algunha das recomendacións do estándar IEC 61131-3 e inclúe editores para configurar o "hardware" e os dispositivos de E/S, estructurar e programar o "software", definir variábeis, depurar e monitorizar os programas, documentar a aplicación, etc.

No referente ao Grafcet, o PL7 permite editar múltiples grafcets conexos que conteñan macroetapas. A estructura das macroexpansións non está limitada a un único grafcet conexo, aínda que si ten que ter unha única etapa de entrada e outra de saída. Non se inclúe ningunha outra extensión sintáctica e só as etapas poden ter identificadores. Cada etapa ten tres accións asociadas de tipos xa predefinidos, non se soportan os tipos de accións do estándar IEC e o algoritmo de interpretación é de tipo SRS. En conclusión, o PL7 e unha aplicación que proporciona múltiples opcións para a programación e monitorización de PLCs Telemecanique. A versión de Grafcet que inclúe permite unha estructura xerárquica de macroetapas pero non soporta os tipos de accións IEC. O programa carece dun simulador e a súa aplicabilidade é limitada.

4.2.5. Visual I/O e Visual PLC

O Visual I/O e o Visual PLC son aplicacións da empresa ArSoft International para o desenvolvemento e execución de programas de control e as súas interfaces. As versións analizadas neste apartado son de demostración, a 4.02 do Visual PLC e a 3.09 do Visual I/O.

4.2.5.1. Compoñentes

O Visual I/O consiste nun ambiente de desenvolvemento de interfaces gráficas Windows que integra un compilador Pascal e que inclúe un conxunto de características adicionais que permiten á súa aplicación en tarefas de control: programación con LD, FBDs ou Grafcet, xestión de alarmas, históricos e receitas, "drivers" de E/S para buses de campo (ModBus, ProfiBus) e PLCs (Unitelway, Omron), etc. O Visual PLC consiste nun módulo softPLC que permite a execución en 'tempo real' dos programas desenvolvidos co Visual I/O. Dende ambos programas poden utilizarse librarías que inclúen funcións para o manexo do tempo, arquivos de texto, dispositivos multimedia, bases de datos, impresión, comunicacións serie, etc.

4.2.5.2. O editor Grafcet

Visual I/O e Visual PLC inclúen un editor Grafcet (Figura 4.10) no que a área de traballo está dividida en filas e columnas e que permite as operacións básicas de edición nun ambiente gráfico: seleccionar, mover, copiar, eliminar, desfacer, zoom, etc. A comprobación do cumprimento da regra de alternancia etapa-transición realízase durante a edición dos modelos, pero durante as probas comprobouse que o editor permitía a edición de estructuras non admitidas polo estándar, quedando baixo a responsabilidade do usuario a edición de Grafcets sintacticamente correctos.

4.2.5.2.1. Estructuras de control

O editor permite a edición de múltiples grafcets conexos sen estructura xerárquica, que inclúan as estructuras de control básicas, ademais de permitir a utilización de etapas (transicións) fontes e sumidoiro. Nas probas comprobouse que o editor non considera coma erro que unha transición sexa á vez fonte e sumidoiro e durante a execución as transicións fonte non se comportaron coma se estiveran sempre validadas. A colocación das etapas iniciais está restrinxida á primeira fila[35] do área de traballo e as seleccións de secuencia e paralelismo poden utilizarse de forma flexíbel, permitíndose a edición de estructuras complexas. Sen embargo tamén poden editarse estructuras incorrectas (Figura 4.10) que non son detectadas durante a análise sintáctica. Os ciclos e saltos de secuencia fanse mediante referencias de salto podendo iniciarse múltiples saltos dende o mesmo punto da secuencia de control.

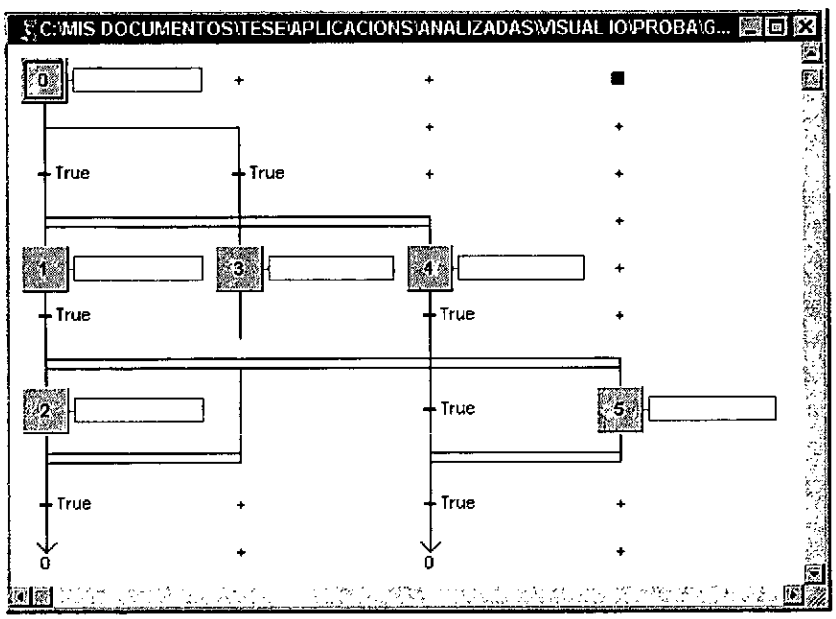


Figura 4.10. Editor Grafcet do Visual I/O e Visual PLC.

4.2.5.2.2. Estructura xerárquica

Os grafcets permitidos polo Visual I/O non teñen estructura xerárquica. O que si permite o programa é modificar o estado de activación das etapas directamente a través das variábeis que almacenan o seu valor. Aínda que isto está prohibido polos estándares IEC, é unha maneira de emular as ordes de forzado. O inconveniente é que non se inclúe soporte á detección de ciclos na xerarquía de forzado nin medios de evitar o forzado múltiple dunha situación.

4.2.5.2.3. Accións e receptividades

O contido de accións e receptividades é codificado mediante a linguaxe Pascal. O código edítase nun editor de texto que comproba a corrección da sintaxe ao tempo que se escribe. As variábeis utilizadas poden ser de calquera dos tipos de datos Pascal e teñen que declararse nunha VPU xunto con todas as outras variábeis do programa: variábeis internas, de entrada, de saída e de estado. Para cada grafcet defínese un conxunto de variábeis que almacenan o valor do estado, o tempo de activación e o indicador de activación (denominado *Start_Step*) de cada

[35] Aínda que despois é posíbel movelas ou copialas a outras filas, co que a restricción deixa de ter senso.

etapa. O Visual I/O permite a modificación do valor destas variábeis dende o código do programa, o que está expresamente prohibido polos estándares IEC.

O programa non soporta directamente os tipos de accións IEC, aínda que algúns poden ser emulados dende o código mediante a utilización das variábeis de estado do grafcet. Por exemplo, pode limitarse ou demorarse a execución dunha acción comprobando o valor do tempo de activación da etapa á que estea asociada (tipos D e L), ou pode executarse un código determinado no momento da activación dunha etapa utilizando a variábel *Start_Step* (tipo P0).

Nas receptividades poden utilizarse os operadores lóxicos e relacionais do Pascal así como chamadas a funcións. Sen embargo o programa non inclúe operadores para a detección de flancos nin para a utilización de temporizadores. Tampouco se inclúe ningún medio de evitar os 'efectos colaterais' que poidan producirse durante a avaliación das receptividades.

4.2.5.2.4. Identificación

O Visual I/O asigna automaticamente identificadores numéricos únicos ás etapas. O usuario non pode modificar estes identificadores nin identificar as transicións. O programa inclúe unha opción de renumeración que actualiza automaticamente os identificadores de etapa para evitar duplicados. Esta opción tamén actualiza correctamente as referencias de salto. O editor non comproba a existencia de duplicados durante a verificación sintáctica do modelo.

4.2.5.2.5. Análise sintáctica

A análise sintáctica faise en dúas fases, o editor comproba o cumprimento da regra de alternancia etapa-transición durante a edición, e a corrección do código de accións e receptividades durante a compilación. Sen embargo a comprobación da corrección sintáctica dos grafcets queda baixo a responsabilidade do usuario, xa que o editor acepta como válidas estructuras non estándar ou identificadores duplicados.

4.2.5.2.6. Simulación e execución

O Visual PLC proporciona as opcións precisas para simular ou executar os programas desenvolvidos co Visual I/O. Os programas compílanse para xerar VPUs que poden cargarse no módulo softPLC do Visual PLC. Este módulo pode estar no mesmo computador ou nun computador diferente e a súa interface ofrece información sobre os módulos cargados e o seu estado. Outras opcións que inclúe o Visual PLC para a simulación e depuración dos programas son, por exemplo, a visualización dos valores das variábeis, a simulación das E/S, a animación gráfica do estado do grafcet, etc. Ademais pódense utilizar os valores das variábeis e animacións do estado do grafcet dende as interfaces de operador desenvolvidas co Visual I/O.

4.2.5.2.7. O algoritmo de interpretación

O algoritmo de interpretación utilizado no motor en tempo real do visual PLC é de tipo SRS, executando as accións tanto en situacións estábeis como inestábeis. Os resultados das probas realizadas para comprobar a resposta do Visual I/O cando varias accións modifican simultaneamente unha mesma variábel (§4.1.2.2) foron os seguintes: o caso (1.a) non puido probarse xa que todas as accións no Visual I/O son continuas; e os casos (1.b), (1.c) e (1.d) non foron considerados coma erróneos polo programa, mudando o valor da variábel de forma non determinista mentres as accións estiveron activas.

4.2.5.3. Conclusións

Visual I/O e Visual PLC son dúas aplicacións implementadas partindo dun ambiente de programación e desenvolvemento de interfaces gráficas Windows utilizando Visual Pascal. A esta base engadíronselle un conxunto de funcións adicionais (programación en LD, FBD e Grafcet, xestión de alarmas, históricos e receitas, etc.) para a súa utilización no desenvolvemento de aplicacións de control en PCs. O conxunto complétase cun módulo softPLC para a execución dos programas e "drivers" de E/S para diferentes buses de campo (ModBus, ProfiBus) e PLCs (Unitelway, Omron).

A versión Grafcet implementada no Visual I/O e o soporte proporcionado polo editor son bastante limitados. Só se permiten as estructuras de control básicas, non se permite unha estructura xerárquica, non se implementan os diferentes tipos de accións IEC e non se poden utilizar eventos nin temporizadores (aínda que si os tempos de activación das etapas) nas condicións. O editor permite a edición de estructuras Grafcet non estándar ou con identificadores duplicados, a interpretación dos modelos é de tipo SRS e non se manexan correctamente os conflictos entre accións. En resume, o Visual PLC aporta unha idea nova como é a de integrar as linguaxes gráficas do IEC nun ambiente de desenvolvemento de aplicacións Windows e proporcionar un módulo softPLC e "drivers" de E/S para a execución dos programas. Sen embargo a versión Grafcet implementada non aporta nada novo e nalgunhas das características comentadas nin sequera cumpre aspectos básicos dos estándares.

4.2.6. AutomGen

AutomGen é unha aplicación da empresa francesa Irai para o desenvolvemento de software de control que pode executarse en PCs ou PLCs. O programa inclúe un editor gráfico multilinguaxe (permite utilizar Gemma, Grafcet, loxigramas, organigramas, LD, FBD, IL e ST), un compilador, un simulador e un editor de pantallas de operador. Para permitir a portabilidade, AutomGen xera durante a compilación un código intermedio que é traducido á linguaxe propia de cada PLC mediante un post-procesador. A versión analizada é a 7.0 de demostración.

4.2.6.1. Compoñentes

A compoñente principal do AutomGen é o *navegador* (Figura 4.11) no que se organizan mediante unha estructura en forma de árbore todos os elementos que forman parte dun proxecto: folios (programas), táboas de símbolos, postprocesadores (configuración "hardware"), documentación e outros elementos relacionados coa depuración e o deseño de pantallas de operador. Dende o navegador están tamén dispoñíbeis opcións para seleccionar o postprocesador a utilizar, acceder á libraría de estructuras predefinidas para as linguaxes gráficas e editar a lista de símbolos do proxecto. A aplicación complétase coa posibilidade de editar e executar a interface gráfica mediante un programa externo denominado IRIS, que permite a animación de obxectos 2D e 3D.

4.2.6.2. O editor Grafcet

O AutomGen non inclúe un editor Grafcet específico, senón que se utiliza o mesmo para todas as linguaxes gráficas, pudendo definirse múltiples 'redes' multilinguaxe en cada folio. Os folios organízanse en filas e columnas e en cada cela pode inserirse un dos múltiples símbolos das linguaxes gráficas soportadas. O programa permite inserir e eliminar filas e columnas, e

soporta as operacións habituais en ambientes gráficos: seleccionar, mover, copiar, eliminar, zoom, etc. Ademais inclúe dúas opcións de grande utilidade durante a edición:

1. *Unha libraría* coas estructuras gráficas máis comúns de cada linguaxe xa predefinidas, de xeito que o usuario só ten que seleccionalas e copialas nos folios.
2. *Un asistente* que permite inserir estructuras complexas mediante a modificación duns poucos parámetros sen ter que editalas símbolo a símbolo.

Un caso particular é a edición de folios Gemma [121], neste caso o folio xa contén un diagrama de estados predefinido e unicamente hai que modificar os estados, liñas de unión e condicións. O programa inclúe unha opción que permite pasar directamente da representación Gemma ao grafcet equivalente.

4.2.6.2.1. Estructuras de control

Poden editarse grafcets que conteñan calquera das estructuras sintácticas básicas así como etapas (transicións) fontes e sumidoiro, macroetapas e ordes de forzado. O uso das estructuras de selección de secuencia e paralelismo é flexíbel, sendo posíbel editar estructuras complexas. Os ciclos e saltos de secuencia poden representarse mediante arcos orientados, e poden iniciarse varios dende un mesmo punto da secuencia de control mediante unha estructura de inicio de selección. O editor non considera coma erro que unha transición sexa simultaneamente fonte e sumidoiro.

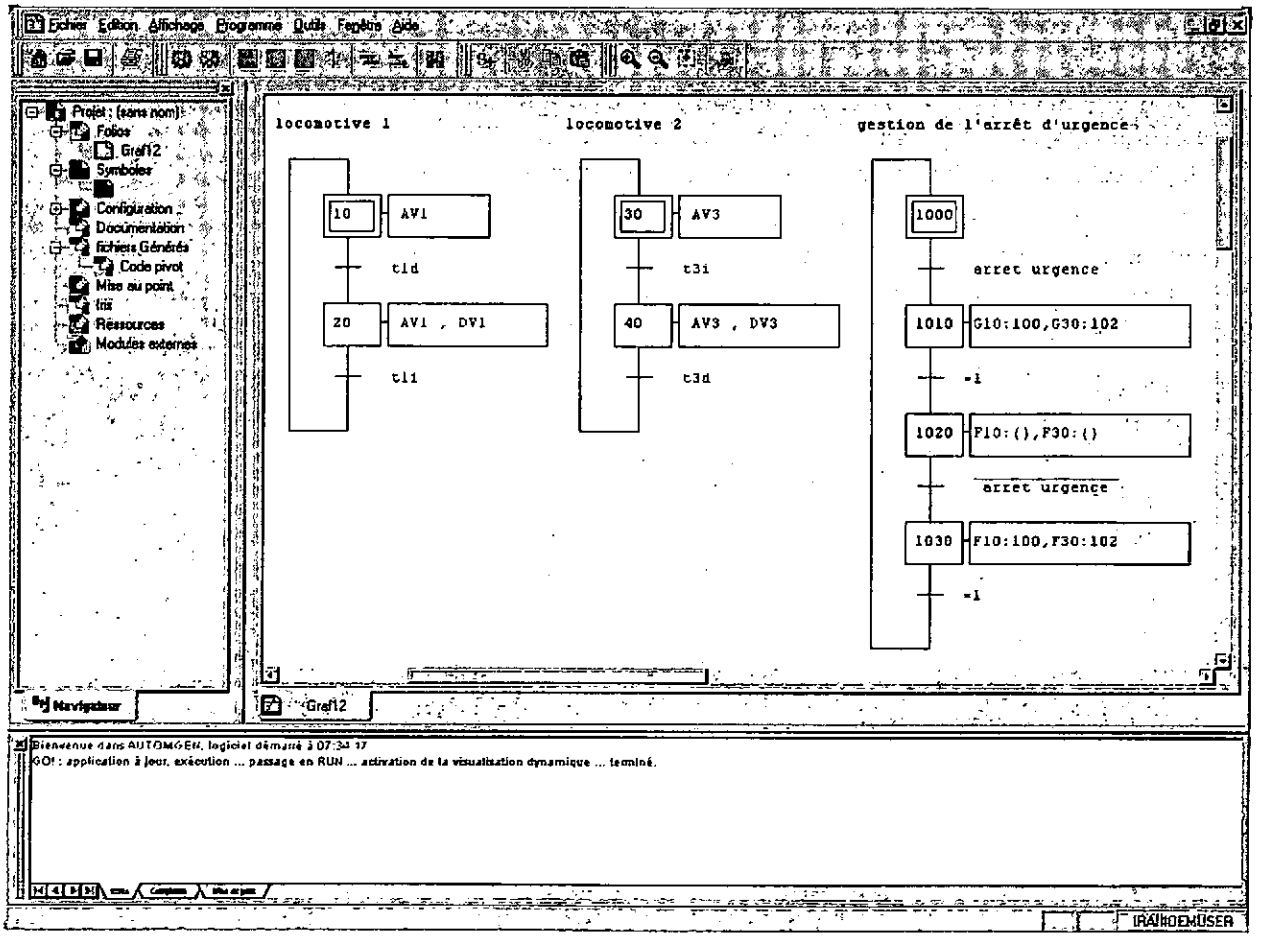


Figura 4.11. Interface do programa Automgen.

4.2.6.2.2. Estructura xerárquica

O AutomGen soporta as dúas estructuras xerárquicas do Grafcet: a estructural e a de forzado, aínda que non sexa posíbel ver ningunha das dúas no navegador do proxecto. A utilización de macroetapas na xerarquía estructural sigue as seguintes regras:

1. Cada macroetapa pode ser utilizada unha única vez na secuencia de control.
2. As macroetapas poden aniñarse.
3. A macroexpansión ten que definirse nun folio diferente e asociarse á macroetapa indicando o seu identificador nas propiedades do folio.
4. A macroexpansión non está limitada a un único grafcet conexo, aínda que só pode conter unha única etapa de entrada e outra de saída.
5. A etapa de entrada ten que usar o identificador 0 ou E_n e a de saída 9999 ou S_n.

Durante a análise comprobouse que o AutomGen detecta durante a compilación as macroetapas sen expansión e as macroexpansións que conteñen máis dunha etapa inicial ou final. Sen embargo aceptou como correctas macroexpansións sen etapa inicial ou final (indicouno cun aviso do compilador), coa etapa inicial e a final en grafcets conexos diferentes, ou macroexpansións con estructuras cíclicas (a etapa final conectada á inicial mediante unha transición).

No referente ás ordes de forzado, o programa non inclúe o concepto de grafcet parcial, mais si permite o agrupamento de múltiples grafcets conexos mediante unha directiva de compilación. Para referenciar un grafcet nas ordes de forzado pode utilizarse o identificador do grupo ao que pertenza, o identificador de calquera das súas etapas ou o identificador do folio no que está definido. O uso correcto dos agrupamentos e ordes de forzado deixase baixo responsabilidade do usuario sendo posíbel, por exemplo, incluír un mesmo grafcet conexo en varios grupos distintos, forzar etapas inexistentes ou definir xerarquías de forzado cíclicas. Ademais do uso estándar das ordes de forzado, AutomGen inclúe dúas extensións:

1. *As ordes de bloqueo*, que son un caso particular das ordes de forzado nas que se bloquea o estado dun grafcet na súa situación actual mentres a orde estea activa.
2. *As ordes de memorización* da situación do grafcet, que permiten almacenar a situación actual dun grafcet para utilizala posteriormente nas ordes de forzado.

A Táboa 4-II resume a sintaxe da directiva de agrupamento e das ordes de forzado en AutomGen.

#G<*id_grupo*>:<*id_graf1*>, ..<*id-grafN*>;	Directiva de agrupamento
F<*id_graf*>:{<*situación*>}	Orde de forzado
F<*id_graf*>:{}	Forzado dunha situación baleira
G<*id_graf*>:<*store*>	Orde de memorización en *store* da situación actual
G<*id_graf*>:<*store*>{<*situación*>}	Orde de memorización en *store* dunha situación específica
F<*id_graf*>:<*store*>	Orde de forzado da situación memorizada en *store*
F<*id_graf*>	Orde de bloqueo

Táboa 4-II. Sintaxe da directiva de agrupamento e ordes de forzado en AutomGen.

4.2.6.2.3. Accións e receptividades

O AutomGen permite utilizar nas accións e receptividades elementos de memoria de tipo booleano (entradas, saídas, bits de etapa Grafcet, bits do sistema, etc.), numérico (variábeis de usuario, contadores, etc.) ou temporizadores, mediante unha sintaxe propia ou dacordo á

sintaxe do IEC 61131-3. O contido das accións pode especificarse mediante sentencias nas linguaxes IL, ST ou mediante unha das sintaxes da Táboa 4-III, que permiten a modificación directa do valor das variábeis. AutomGen implementa os tipos de accións IEC seguintes[36]: N, R, S, P1 e P0, ademais inclúe accións condicionais que xunto cos temporizadores permiten emular as accións temporizadas L, D e DS do estándar.

En canto ás receptividades, pode utilizarse nelas calquera elemento de memoria (incluídas as saídas); os operadores booleanos AND (.), OR (+) e NOT (/); os de comparación; eventos (↑,↓) en variábeis e expresións; e temporizacións, cunha das sintaxes seguintes:

<duración>/<var_id>/T_n
T_n/<var_id>/<duración>
<duración>/<var_id>

A propia sintaxe das receptividades impide a posibilidade de introducir código que produza 'efectos colaterais' durante a súa avaliación. Os únicos valores que se modifican son os dos temporizadores iniciados na receptividade.

Cualificador	Tipo de variábel	Efecto
<var_name>	booleana, temporizador	Mantén o valor a 1 durante a activación da etapa
N *<var_name>*	booleana, temporizador	Mantén o valor a 0 durante a activación da etapa
R *<var_name>*	todas	Posta a 0 do valor (memorizado)
S *<var_name>*	booleana, temporizador	Posta a 1 do valor (memorizado)
I *<var_name>*	booleana, temporizador	Invirte o valor (unha vez por ciclo durante a activación da etapa)
[+|-] *<var_name>*	numérica, contador	Incrementa/decrementa en 1 o valor (unha vez por ciclo durante a activación da etapa)

Táboa 4-III. Sintaxe das accións que modifican directamente o valor dunha variábel no AutomGen.

4.2.6.2.4. Identificación

No AutomGen só poden asignarse identificadores numéricos ás etapas (excepción feita das etapas de entrada e saída das macroexpansións que teñen unha sintaxe específica). O programa non comproba a existencia de duplicados durante a edición e inclúe unha opción para numerar automaticamente todas ou parte das etapas dun grafcet (sen actualizar os contidos de accións e receptividades). A utilización de identificadores alfanuméricos, as etapas sen identificador, as variábeis de etapas inexistentes e os identificadores duplicados son detectados durante a compilación.

4.2.6.2.5. Análise sintáctica

A verificación sintáctica realízase durante a compilación dos folios. A única asistencia previa á compilación que proporciona o programa é durante a edición de accións e receptividades. Esta realízase mediante un cadro de diálogo composto de dúas partes: unha na que se edita o código que o programa marca con diferentes cores a medida que comproba a súa corrección sintáctica, e outra na que se proporciona acceso á sintaxe dos elementos de memoria e accións así como aos símbolos do proxecto.

[36] Na documentación indícase que se soportan tamén os tipos L, D, SD, DS e SL, mais a versión do programa analizada non permitiu utilizar estes cualificadores.

Durante as probas comprobouse que o compilador do AutomGen acepta como correctas as macroexpansións sen etapa inicial ou final (indícao cun aviso), coa etapa inicial e a final en grafcets conexos diferentes ou con estructuras cíclicas (a etapa final conectada á inicial mediante unha transición); permite o agrupamento dun grafcet conexo en varios grupos distintos, o uso de etapas inexistentes nas directivas de agrupamento e nas ordes de forzado, a existencia de varias directivas de agrupamento cun mesmo identificador de grupo e non detecta as xerarquías de forzado cíclicas nin os forzados múltiples.

4.2.6.2.6. Simulación e execución

O AutomGen permite a simulación e a monitorización en tempo real dos programas. O editor inclúe opcións para compilar, descargar e executar de forma continua ou paso a paso os programas, visualizar e forzar o valor das variábeis, animar graficamente nos folios os valores de etapas, variábeis e temporizadores, simular as entradas, etc. Co postprocesador adecuado pode executarse o código directamente no PC ou descargalo nun PLC (hai dispoñíbeis postprocesadores para un grande número de fabricantes: Omron, Telemecanique, Siemens, etc.). Dende o postprocesador PC poden utilizarse "drivers" para diferentes sistemas de E/S (tarxetas de adquisición de datos, buses de campo, etc.) e tamén pode configurarse o modo de funcionamento (periódico ou continuo) e o tempo máximo de ciclo permitido.

4.2.6.2.7. O algoritmo de interpretación

O algoritmo de interpretación utilizado no postprocesador PC do AutomGem é de tipo ARS sen detección de ciclos estacionarios (cando o grafcet entra nun ciclo estacionario o comportamento do sistema deixa de ser determinista). Durante as evolucións internas do modelo os eventos externos foron considerados coma eventos na escala interna e os eventos internos tratáronse internamente (§4.1.2.1, caso 1.a). Ademais comprobouse que a pesar de utilizar un algoritmo ARS, todas as accións, ordes de forzado e bloqueo son executadas tanto nas situacións estábeis como nas inestábeis, o que significa que o programa non diferencia entre accións internas e externas.

Tamén se atoparon algúns problemas durante a comprobación da aplicación da xerarquía de forzado (§4.1.2.3):

1. No grafcet da Figura 3.37, partindo da situación {1, 10, 20, 30}, ao producirse o evento $\uparrow a$ a evolución foi {1, 10, 20, 30}→{2, 11, 21, 32}, o que indica que a evolución estructural (o franqueamento da transición 10) tivo prioridade sobre a aplicación da orde de forzado da etapa 21.
2. No mesmo grafcet, cando se produce o evento $\uparrow b$ na situación {2, 11, 21, 32}, a evolución foi {2, 11, 21, 32}→{3, 11, 21, 32}→{3, 11, 22, 32}, do que se deduce que ao deixar de estar activa a orde de forzado da etapa 2 e activarse a etapa 3 pode franquearse a transición 21, mais isto tamén desactiva a orde de forzado da etapa 21 o que provocaría o franqueamento da transición 32, o cal non acontece.

En resume, no AutomGen a aplicación das ordes de forzado non ten prioridade sobre as evolucións estructurais e non se aplica recursivamente a xerarquía de forzado.

En canto aos conflictos entre accións, AutomGen define unha táboa de incompatibilidades que se reproducen na Táboa 4-IV. Cando o programa detecta algunha destas incompatibilidades durante a compilación indícao mediante un aviso. Polas probas realizadas dedúcese que o programa da o aviso mesmo se non hai posibilidade de que as accións conflictivas estean

activas simultaneamente, polo que é responsabilidade do usuario determinar se o posíbel conflicto existe realmente ou non.

	Continua	N	S	R	I
Continua		X	X	X	X
N	X		X	X	X
S	X	X			
R	X	X			
I	X	X			

Táboa 4-IV. Incompatibilidades entre accións definidas por AutomGen

Esta incompatibilidade non afecta ás probas realizadas (§4.1.2.2) para analizar a resposta do programa cando varias accións modifican simultaneamente unha mesma variábel, xa que isto pode acontecer entre accións que a priori son compatíbeis. Os resultados para o AutomGen foron os seguintes: no caso (1.a) a saída mantivo o valor 1, o que indica que a acción R non é prioritaria sobre as demais en contra do definido no estándar IEC 61131-3; o caso (1.b) non foi considerado coma un erro e a saída tomou o valor resultado da disxunción lóxica das accións activas; e, por último, os casos (1.c) e (1.d) tampouco foron considerados coma erros, e a variábel tomou diferentes valores mentres as accións estiveron activas.

4.2.6.3. Conclusións

O AutomGen é unha aplicación de desenvolvemento de software de control para PC e unha ampla gama de PLCs. O programa soporta un grande número de linguaxes: Gemma, Grafcet, loxigramas, organigramas, LD, FBD, IL e ST, e inclúe opcións para a configuración do "hardware" e dos dispositivos de E/S, a edición, compilación, descarga, control e depuración en tempo real da execución dos programas, a animación das linguaxes gráficas, a monitorización e forzado de variábeis durante a depuración, etc. Ademais tamén pode editarse unha interface gráfica para a supervisión do proceso que pode executarse mediante un programa externo denominado IRIS.

En canto ao Grafcet, o AutomGen é moi completo soportando case todas as características definidas polos estándares IEC. Entre os aspectos negativos atopados durante as probas poden citarse: só as etapas poden ter identificadores (numéricos); permítense macroexpansións sen etapa inicial ou final e con estructuras cíclicas; non se inclúe o concepto de grafcet parcial, aínda que se permite o agrupamento de grafcets conexos; as ordes de forzado poden conter etapas inexistentes; a xerarquía de forzado pode conter ciclos; non se soportan directamente as accións temporizadas do IEC; o algoritmo de interpretación é de tipo ARS mais non detecta ciclos estacionarios e a detección de conflictos entre accións é limitada. En resume, o AutomGen é unha excelente aplicación que destaca polo número de linguaxes que inclúe, a gama de PLCs e sistemas de E/S que soporta e as capacidades de animación gráfica durante a simulación dos programas. Implementa case todas as características do Grafcet e é o único dos programas analizados que ofrece soporte ao estándar Gemma.

4.2.7. Actwin

ActWin é unha aplicación da empresa sueca Actron para a implementación de proxectos de automatización con PLCs da empresa Hitachi dende un PC con sistema operativo Windows. O programa inclúe opcións para configurar os PLCs, configurar as comunicacións e dispositivos de E/S, editar o diccionario, os programas e a documentación do proxecto, cargar o proxecto no

PLC e monitorizar o seu funcionamento en tempo real. O ambiente de desenvolvemento sigue en parte as especificacións do estándar IEC 61131-3 e as linguaxes permitidas para a programación son todas as do estándar menos a ST. A aplicación tamén inclúe un "driver" softPLC que permite depurar e executar os programas no propio PC sen necesidade de dispor dun PLC. A versión do ActWin analizada é a 3.26 de demostración.

4.2.7.1. Compoñentes

Tres son as principais compoñentes da arquitectura do ActWin:

1. *O ambiente de desenvolvemento* (Figura 4.12), que é unha aplicación Windows que sigue as recomendacións do estándar IEC 61131-3 e proporciona as opcións precisas para a configuración, programación, documentación, depuración e monitorización do proxecto. Dentro deste ambiente as principais opcións están dispoñíbeis a través de tres editores:
   a. *O editor do proxecto,* no que se organizan mediante unha estructura en forma de árbore os diferentes elementos que compoñen un proxecto: configuración do "hardware" e dispositivos de E/S, POUs, táboas de símbolos, librarías de FBs e outras opcións relacionadas coa impresión e a monitorización do proxecto.
   b. *O editor de símbolos* dende o que se editan as propiedades dos símbolos utilizados no proxecto.
   c. *Os editores de programas* nos que se editan as POUs utilizando as linguaxes IL, LD, FBD e SFC.

   O ambiente complétase cun conxunto de diálogos que facilitan a edición da configuración "hardware", a asignación de símbolos a puntos de E/S, a escolla de instruccións, operadores e símbolos durante a edición dos programas, etc.

2. *Os "drivers" PLC*, que implementan para os PLCs soportados as funcións que permiten a súa configuración, a dos dispositivos de E/S e comunicacións, a carga e descarga de programas e símbolos, a monitorización dos programas, etc. Ademais dos "drivers" das diferentes versións de PLCs Hitachi, ActWin inclúe un "driver" softPLC co que poden executarse os programas no PC e emular o sistema de control sen necesidade de dispor dun PLC. Este "driver" inclúe características como: a posibilidade de traballar como "master" con distintos buses de campo (Profibus-DP, Interbus-S, CANOpen ou DeviceNet); a monitorización e forzado de variábeis en tempo real; ou a modificación "on-line" dos programas.
3. *As ferramentas para a programación de "drivers"*, orientadas a aqueles usuarios avanzados que queiran desenvolver "drivers" para dar soporte a novos PLCs.

4.2.7.2. O editor Grafcet

O editor do ActWin (Figura 4.12) está orientado á edición de múltiples redes SFC cíclicas que conteñan unha única etapa inicial. Ao comezar a edición, no área de traballo aparece unha rede mínima composta por unha etapa inicial, unha transición e un ciclo. Durante a edición só está permitido inserir novos elementos en puntos concretos do SFC nos que se manteña a súa corrección sintáctica. A cada rede poden asociárselle dúas condicións para controlar a súa execución:

1. *A condición de activación*, que mentres sexa falsa 'conxela' o estado da rede.
2. *A condición de reinicio,* que ao activarse devolve a rede á súa situación inicial.

O editor soporta as opcións de edición máis comúns nos ambientes gráficos: inserir, mover, copiar e eliminar redes ou elementos das redes, zoom, etc. e inclúe unha opción que permite traducir automaticamente os SFCs a LD.

4.2.7.2.1. Estructuras de control

O ActWin unicamente permite a utilización dos elementos sintácticos básicos do Grafcet. As redes SFC editadas teñen que ter unha estructura cíclica e non poden conter máis dunha etapa inicial. As estructuras de selección de secuencia e paralelismo non poden ser utilizadas de forma flexíbel, permitíndose unicamente a edición de estructuras completas. Poden inserirse tamén ciclos (mediante referencias de salto) e saltos de etapa, sendo posíbel iniciar múltiples ciclos ou saltos dende un mesmo punto da secuencia de control.

4.2.7.2.2. Estructura xerárquica

O editor non permite definir unha estructura xerárquica entre SFCs. A estructuración do proxecto realízase a nivel de POU, seguindo as recomendacións do estándar IEC 61131-3.

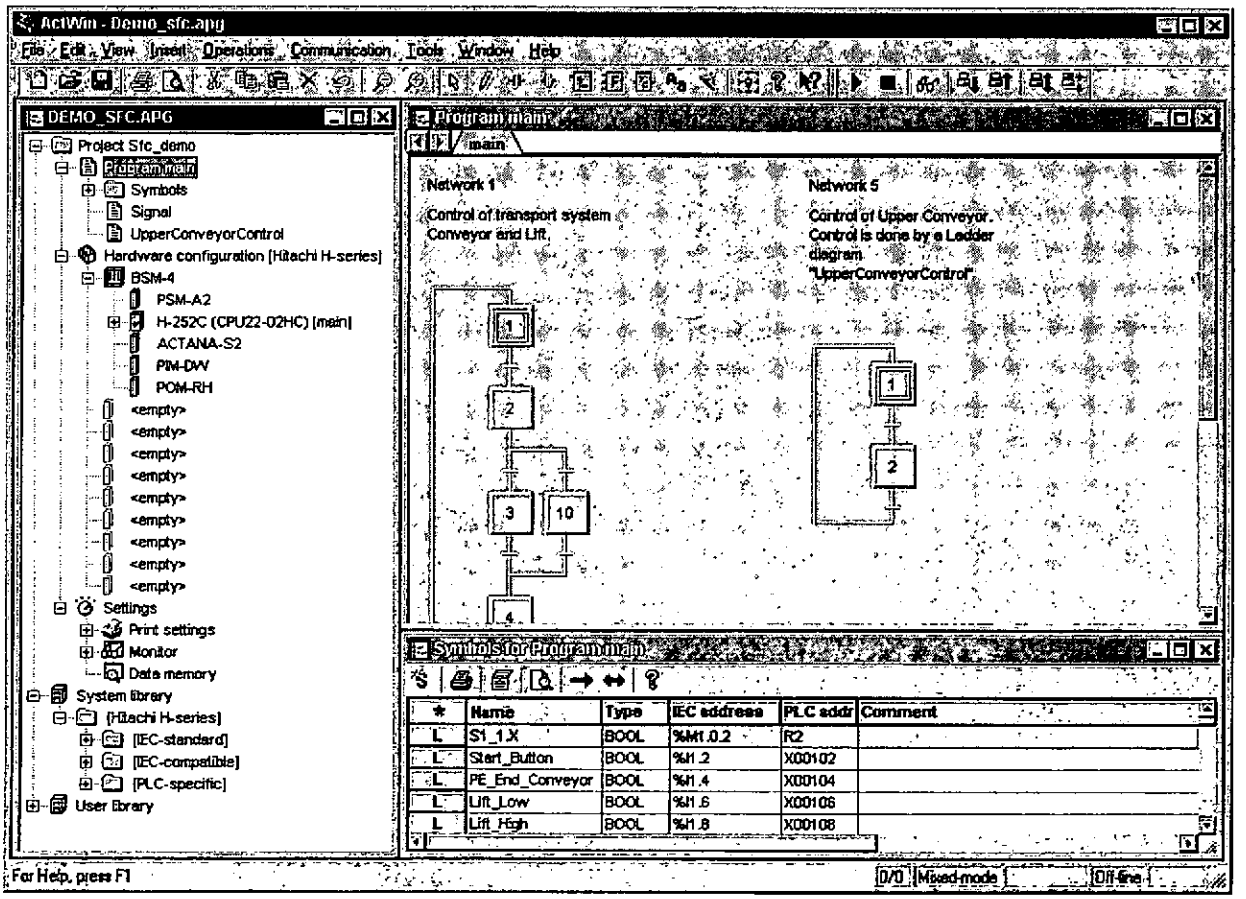


Figura 4.12. Interface da aplicación ActWin.

4.2.7.2.3. Accións e receptividades

O ActWin soporta todos os tipos de datos básicos e formatos de direccionamento definidos no estándar IEC 61131-3. O programa permite escoller entre traballar utilizando a nomenclatura IEC ou a específica de cada PLC. Cada etapa pode conter múltiples accións que poden ser de calquera dos tipos IEC: N, R, S, P, L, D, DS, SD e SL. As accións poden ter

asociado un símbolo booleano, que será modificado durante a activación da etapa, ou ser programadas mediante a linguaxe LD. O mesmo pode dicirse das receptividades, que poden ter asociado un símbolo booleano ou unha condición lóxica en LD. Nos diagramas LD poden usarse os FBs predefinidos que inclúen temporizadores e contadores. O programa non impide a edición de receptividades que produzan 'efectos colaterais' durante a súa avaliación.

4.2.7.2.4. Identificación

O ActWin só permite asignar identificadores alfanuméricos ás etapas e ás accións ou condicións LD. Os identificadores poden coincidir con calquera outro símbolo xa que, para evitar duplicados, a aplicación define internamente os símbolos *<step_id>.X* para as etapas e *<action_id>.Q* para as accións. Por defecto utilízanse para as etapas identificadores co formato S_n_x, sendo n o identificador da rede SFC e x o da etapa. O programa actualiza automaticamente os identificadores e referencias cada vez que se insire ou elimina unha nova rede ou etapa e controla a existencia de duplicados durante a edición. Cada programa ten a súa propia táboa de símbolos, polo que os identificadores poden repetirse en SFCs diferentes.

4.2.7.2.5. Análise sintáctica

O propio proceso de edición do ActWin impide que se editen estructuras SFC incorrectas, que se utilicen identificadores duplicados ou se asignen os símbolos a puntos de E/S incorrectamente. Ademais o programa inclúe tamén unha opción de verificación sintáctica que realiza a comprobación da corrección do proxecto antes de descargalo no PLC.

4.2.7.2.6. Simulación e execución

A aplicación inclúe as opcións precisas para a carga e descarga de proxectos no PLC, o intercambio de información de configuración e de estado, a monitorización e forzado dos valores das variábeis, a animación dos elementos gráficos nos diagramas LD, FBD e SFC, a modificación "on-line" dos programas, etc. Ademais a dispoñibilidade dun "driver" softPLC permite executar os programas no propio PC simulando os dispositivos de E/S, o que permite depurar os programas sen necesidade de dispor dun PLC.

4.2.7.2.7. O algoritmo de interpretación

O algoritmo de interpretación do "driver" softPLC do ActWin é de tipo SRS. A resposta do ActWin, nos diferentes casos probados cando varias accións modifican simultaneamente unha mesma variábel (§4.1.2.2) foi a seguinte: no caso (1.a) a saída mantivo o valor 0, o que indica que a acción R tivo prioridade sobre as demais dacordo ao definido no estándar IEC 61131-3; o caso (1.b) non foi considerado coma un erro e a saída tomou o valor resultado da disxunción lóxica das accións activas; e, por último, os casos (1.c) e (1.d) tampouco foron considerados coma erros, e á variábel asignáronselle valores de forma errática mentres as accións estiveron activas.

4.2.7.3. Conclusións

O ActWin é unha aplicación que sigue as recomendacións do estándar IEC 61131-3 para a programación de PLCs Hitachi dende PCs con Windows. O programa inclúe opcións para a configuración "hardware" dos PLCs, a definición de símbolos e a súa asignación a puntos de E/S, a edición e verificación dos programas, a descarga dos proxectos nos PLCs, o control e monitorización da súa execución, a visualización e forzado de variábeis durante a depuración,

etc. A aplicación completase cun "driver" softPLC que permite executar os programas no propio PC e ferramentas para o desenvolvemento de "drivers" para soportar novos PLCs.

Dende o punto de vista do Grafcet, o ActWin implementa o SFC dacordo ao definido no estándar IEC 61131-3. Polo tanto só permite utilizar as estructuras de control básicas, non inclúe soporte xerárquico, o algoritmo de interpretación é de tipo SRS e non manexa correctamente os conflictos entre accións. En resume, no ActWin destaca o fácil que resulta a edición dos programas e configuración dos símbolos gracias aos diálogos que guían ao usuario, e o alto grao de conformidade co estándar IEC 61131-3. Ademais a aplicabilidade da aplicación non se limita unicamente á programación de autómatas Hitachi, xa que o "driver" softPLC pode configurarse como "master" de diferentes buses de campo, o que permite realizar o control directamente dende o PC.

4.2.8. WinGrafcet

O WinGrafcet é unha aplicación Windows desenvolvida polo Centro Rexional de Documentación Pedagóxica do Languedoc-Roussillon en colaboración co Centro de Formación Tecnolóxica de Montpellier, para o desenvolvemento de programas de control secuenciais mediante Grafcet. A versión analizada é a versión 1.01 de demostración.

4.2.8.1. Compoñentes

A aplicación está formada por un editor gráfico, un verificador sintáctico, un simulador e un monitor en tempo real compatíbel con múltiples interfaces de E/S (AM1, AM2, IP16, Approtech, Technologie Service, Ipocaen, etc.).

4.2.8.2. O editor Grafcet

O editor Grafcet do WinGrafcet (Figura 4.13) está orientado á edición de múltiples grafcets conexos e permite as operacións básicas de edición nun ambiente gráfico como: seleccionar, mover, copiar, eliminar, zoom, etc. A área de traballo divídese en filas e columnas, podendo inserirse en cada cela un dos elementos sintácticos básicos do Grafcet. O editor inclúe unha opción para detectar e eliminar as estructuras non conexas antes de realizar a verificación sintáctica, e os símbolos utilizados poden verse en dous niveis de descrición diferentes.

4.2.8.2.1. Estructuras de control

O editor facilita a edición de múltiples grafcets conexos que conteñan calquera dos elementos sintácticos básicos do Grafcet. Non se inclúe ningunha das extensións sintácticas nin soporte á estructura xerárquica. Non está permitido que as liñas se crucen e as estructuras de selección de secuencia e paralelismo non poden utilizarse de forma flexíbel, o que impide a edición de grafcets complexos. Poden utilizarse ciclos e saltos de secuencia que son representados mediante referencias de salto. Está permitido iniciar múltiples saltos ou ciclos dende un mesmo punto da secuencia de control.

4.2.8.2.2. Estructura xerárquica

O WinGrafcet non inclúe soporte á estructura xerárquica do Grafcet.

4.2.8.2.3. Accións e receptividades

O WinGrafcet define diferentes elementos de memoria que poden ser utilizados en accións e receptividades: entradas (*e0-e11*), saídas (*s0-s7*), variábeis internas (*v0-v99*), temporizadores

(*t0-t19*) e contadores (*c0-c19*). Os nomes destes elementos están predefinidos e non é posíbel modificalos. A cantidade de elementos dispoñíbeis de cada tipo dependerá da interface de E/S da que se dispoña, aspecto que é controlado polo programa durante a verificación sintáctica.

O editor permite unicamente accións continuas cunha das sintaxes da Táboa 4-V. Nas receptividades poden utilizarse os valores de entradas, variábeis, contadores, temporizadores, e o estado das etapas relacionados mediante operadores booleanos, de comparación e de detección de flancos. O intento de modificación dunha saída dende unha receptividade é detectado polo programa durante a verificación sintáctica impedíndose así os 'efectos colaterais'.

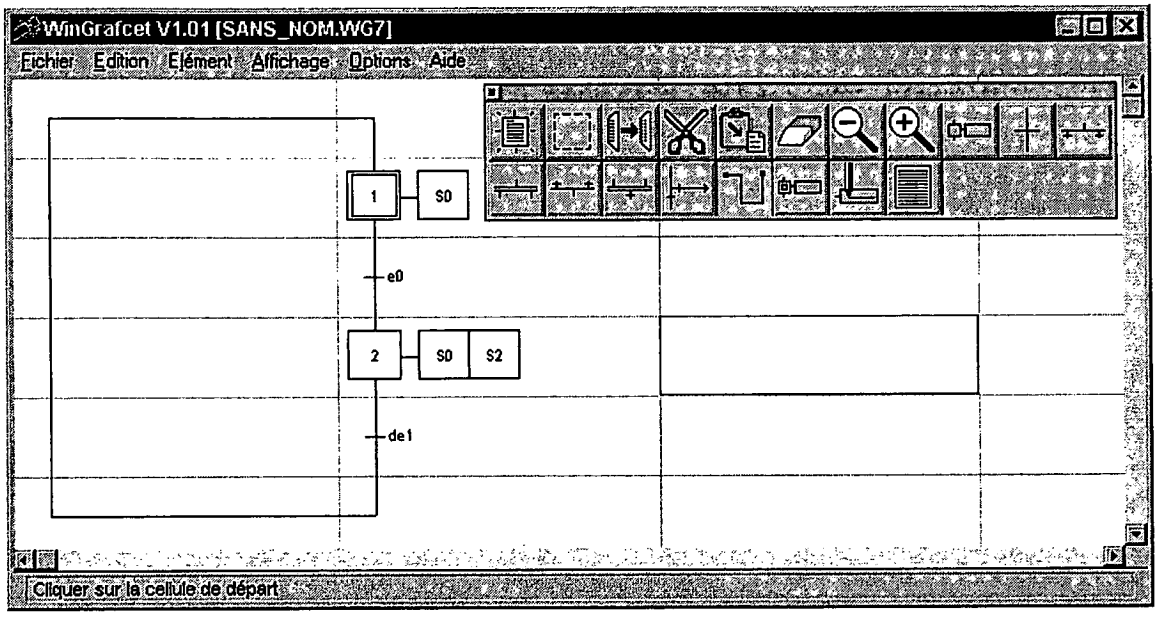


Figura 4.13. Interface do editor Grafcet do WinGrafcet.

S_i	Activar unha saída booleana
C_i(valor)	Iniciar un contador
C_i++, C_i--	Incrementar/decrementar un contador
V_i(valor)	Iniciar unha variábel interna
T_i	Iniciar un temporizador

Táboa 4-V. Sintaxe das accións en WinGrafcet

4.2.8.2.4. Identificación

No WinGrafcet só poden asignarse identificadores numéricos ás etapas. O programa inclúe unha opción de renumeración automática que pode ser utilizada para evitar identificadores duplicados (esta opción non actualiza os contidos das receptividades). Tampouco se renumeran automaticamente as etapas ao copiar nin se controla a existencia de duplicados durante a edición, aínda que si durante a verificación sintáctica.

4.2.8.2.5. Análise sintáctica

O editor incorpora un verificador sintáctico para realizar a comprobación da corrección do modelo Grafcet. Non se realiza ningunha comprobación durante a edición.

4.2.8.2.6. Simulación e execución

O WinGrafcet inclúe un simulador para a depuración dos programas. As evolucións son controladas polo usuario a través dunha interface na que se simulan as entradas mediante botóns e as saídas mediante indicadores luminosos. Durante a simulación pode consultarse a información sobre a situación do modelo, ver os valores das variábeis internas e modificar a velocidade da simulación.

A execución dos programas faise nun monitor en tempo real que permite conectar o PC ao proceso mediante diferentes interfaces de E/S (AM1, AM2, IP16, Approtech, Technologie Service, Ipocaen). Na versión de demostración analizada a opción de monitorización en tempo real está desactivada, polo que non foi posíbel comprobar o seu funcionamento.

4.2.8.2.7. O algoritmo de interpretación

O algoritmo de interpretación do WinGrafcet é de tipo SRS, executándose as accións tanto en situacións estábeis como inestábeis. Debido a que na versión de demostración a opción de simulación só permite simular un exemplo que ven co programa, non foi posíbel realizar as probas para comprobar a resposta do programa cando varias accións modifican simultaneamente unha mesma variábel.

4.2.8.3. Conclusións

O WinGrafcet é unha aplicación para a edición de programas de control secuencial mediante Grafcet. Está formado por un editor gráfico, un verificador sintáctico, un simulador e un monitor en tempo real que permite realizar o control co PC conectado directamente ao proceso mediante diferentes interfaces de E/S. Os Grafcets editados non incorporan ningunha das extensións sintácticas nin inclúen soporte á estructuración xerárquica. Só se permiten accións continuas cunha sintaxe propia e o algoritmo de interpretación é de tipo SRS. En resume, o WinGrafcet é unha aplicación útil para o ensino e o control mediante PC de procesos simples, que implementa unha versión básica do Grafcet na que unicamente poden utilizarse estructuras básicas e accións continuas.

4.2.9. Graf7-C

O Graf7-C é unha aplicación realizada con fines educativos polo Centre Collégial de Développement de Matériel Didactique do Canada, para o desenvolvemento de programas de control mediante o uso conxunto do Grafcet e a linguaxe C. A versión analizada é a 2.1 para o sistema operativo DOS.

4.2.9.1. Compoñentes

A aplicación está formada por un editor, un analizador sintáctico, un compilador, un traductor, un simulador e un monitor en tempo real.

4.2.9.2. O editor Grafcet

O editor Grafcet do Graf7-C (Figura 4.14) traballa en modo texto e soporta as operacións básicas de edición: seleccionar, mover, copiar, eliminar, zoom, etc. Permite a edición de múltiples grafcets conexos distribuídos en catro páxinas de 255 filas por 255 columnas, reservando as filas impares para as etapas e as pares para as transicións. Os grafcets poden visualizarse en tres niveis diferentes de detalle: documentación, programación e código C, e a corrección sintáctica das estructuras editadas é comprobada durante a edición.

4.2.9.2.1. Estructuras de control

Para a edición dos grafcets poden utilizarse todas as estructuras de control básicas e algunha das extensións sintácticas: etapas (transicións) fontes e sumidoiro, macroetapas e ordes de forzado. O uso das estructuras de selección de secuencia e paralelismo é flexíbel, pudendo editarse estructuras complexas. Os ciclos e saltos de secuencia poden representarse mediante arcos ou mediante referencias de salto, e poden iniciarse varios ciclos ou saltos dende un mesmo punto da secuencia de control. O editor non considera coma erro que unha transición sexa simultaneamente fonte e sumidoiro. Ademais o editor permite a utilización doutros elementos sintácticos non estándar:

1. *As tarefas*, que son unha alternativa ás macroetapas en PLCs programados mediante diagramas de contactos.
2. *As ordes de bloqueo*, que son un caso particular das ordes de forzado que bloquean o estado dun grafcet na situación actual mentres a orde estea activa.

4.2.9.2.2. Estructura xerárquica

O Graf7-C soporta as dúas estructuras xerárquicas do Grafcet: a formada polas macroetapas e a formada polas ordes de forzado. As macroetapas son utilizadas unha única vez na secuencia de control, poden aniñarse, está permitida a asociación de accións tanto ás etapas de inicio e fin da macroexpansión como ás macroetapas (característica non estándar), e as macroexpansións teñen que comezar por unha única etapa de inicio e rematar por unha única etapa final. Sen embargo, durante as probas detectouse que o editor unicamente comproba que os identificadores das macroetapas (M_x), das etapas de inicio (E_x) e de fin (S_x) da macroexpansión sexan únicos, sendo posíbel a edición de macroexpansións incorrectas (sen etapa de inicio ou fin), e a existencia de macroetapas sen macroexpansión ou viceversa.

En canto ás ordes de forzado, o editor non inclúe o concepto de grafcet parcial polo que estas limítanse ao forzado de grafcets conexos. Para poder forzar un grafcet conexo é preciso asociarlle previamente un identificador que será utilizado nas ordes de forzado. O editor deixa baixo a responsabilidade do usuario a coherencia das ordes de forzado, sendo posíbel na práctica editar ordes de forzado que forcen etapas inexistentes e definir estructuras de forzado que conteñan ciclos ou provoquen situacións de forzado múltiple.

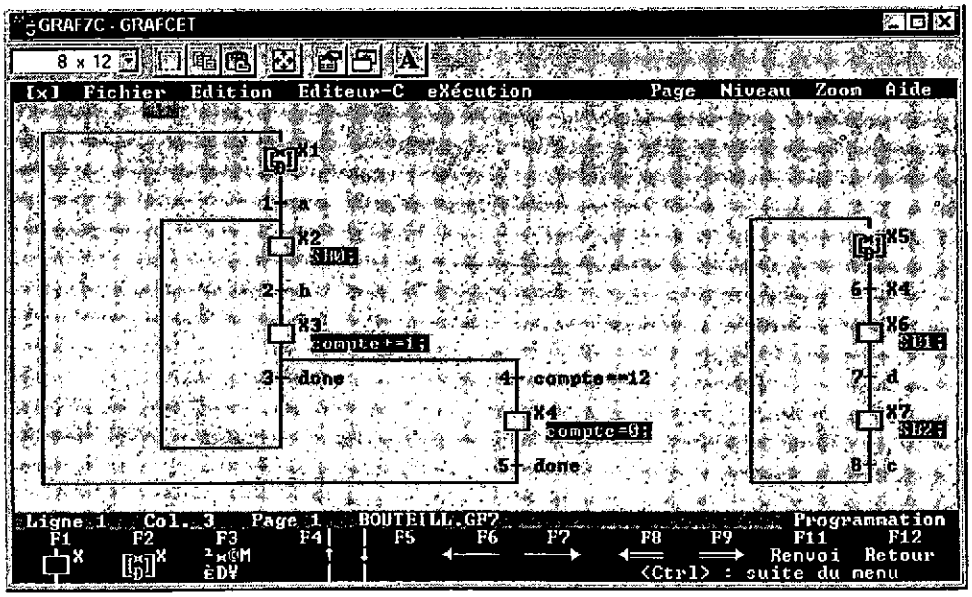


Figura 4.14. Interface do editor Grafcet do Graf7C.

4.2.9.2.3. Accións e receptividades

No Graf7C o código de accións e receptividades é especificado utilizando unha versión da linguaxe C, denominada C7, que foi implementada partindo dun subconxunto do C estándar ao que se lle engadiron algunhas características novas, como:

1. Un control mais estricto dos tipos de datos.
2. Novos operadores para representar aspectos específicos do Grafcet: ordes de forzado, pulsos, temporizadores, eventos, etc.
3. Unha libraría moi variada de funcións para a visualización, acceso a arquivos, programación de interfaces de E/S, etc.

Cada acción está composta por un conxunto de sentencias C7 consideradas internamente como o código dunha función C de tipo *void* (función que non devolve ningún valor). Cada sentencia pode ser polo tanto unha declaración de variábel local, unha sentencia C (o que inclúe: bucles, seleccións, saltos, chamadas a funcións, etc.) ou unha acción dalgún dos tipos indicados na Táboa 4-VI. Destas accións as continuas e memorizadas poden utilizarse unicamente con variábeis booleanas, as condicionais defínense mediante a instrucción *if/else* do C e as impulsionais utilizando a función *pulse* implementada polo C7 (esta función devolve o valor *true* mentres o código da acción non sexa executado). A utilización de temporizadores nas accións condicionais permite editar tamén accións demoradas e limitadas no tempo. A sintaxe C7 para os temporizadores pode ser unha das seguintes:

$$T/<t_1>s/<X_n>$$
$$T/<X_n>/<t_2>s$$
$$T/<t_1>s/<X_n>/<t_2>s$$
$$T/<t_1>s/[\uparrow|\downarrow]<expresión>$$

polo que as accións demoradas e limitadas representaríanse como:

if (T/1s/X_i) *acción*; **// acción retardada**
if (T/X_i/1s) *acción*; **// acción limitada**
if (T/1s/X_i/2s) *acción*; **// acción retardada e limitada**

Sen embargo durante as probas realizadas, os temporizadores que inclúen un tempo t_2 non funcionaron correctamente, polo que foi preciso representar as accións limitadas da forma seguinte:

if (!(T/1s/X_i)) *acción*; **// acción limitada**
if (T/1s/X_i && !(T/2s/X_i)) *acción*; **// acción retardada e limitada**

Continuas	<*variábel*>;
Memorizadas	<*variábel*> = [1|0];
Condicionais	**if** (<*condición*>) <*acción T*> [**else** <*acción F*>]
Impulsionais	**if** (pulse()) <*acción*> <*variábel*> = pulse(); <*contador*> [+|-]= pulse();
Ordes de forzado	F/<*subgrafcet*>:(<*situación*>)
Ordes de bloqueo	F/<*subgrafcet*>:(*)

Táboa 4-VI. Sintaxe das accións no Graf7-C

As receptividades no Graf7-C son condicións lóxicas escritas en C7, polo que poden utilizarse os operadores lóxicos e relacionais do C, chamadas a funcións, temporizadores, eventos e unha macro denominada *done*, que é certa cando as accións asociadas ás etapas previas á transición foron xa executadas. O Graf7-C non impide que durante a avaliación das condicións poidan producirse 'efectos colaterais'.

4.2.9.2.4. Identificación

O Graf7-C permite asignar identificadores ás etapas, transicións e grafcets conexos utilizados na xerarquía de forzado. Os formatos dos identificadores son: X_n para as etapas, M_n para as macroetapas, E_n e S_n para as etapas de inicio e fin de macroexpansión, identificadores numéricos para as transicións e alfanuméricos para os grafcets conexos. O editor asigna os identificadores de etapa e transición automaticamente, controla que non existan duplicados e renumera automaticamente as etapas e transicións despois dunha operación de copia, aínda que esta característica non actualiza os contidos de accións e receptividades e non funciona correctamente coas referencias de salto.

4.2.9.2.5. Análise sintáctica

O Graf7-C realiza a análise sintáctica en dúas fases, durante a edición só poden inserirse símbolos e realizar conexións que cumpran a regra de alternancia etapa-transición, e impídese o uso de identificadores duplicados. A detección de erros sintácticos no contido de accións e receptividades realízase durante a compilación do grafcet. Durante as probas comprobouse que o compilador non detecta o uso incorrecto de macroetapas (macroexpansións sen etapa de inicio ou fin, macroetapas sen macroexpansión ou viceversa), erros nas ordes de forzado (forzado de etapas inexistentes, autoforzado dun grafcet), ou a existencia de arcos non conectados.

4.2.9.2.6. Simulación e execución

O Graf7-C permite tanto a simulación como a execución dos grafcets co PC directamente conectado ao proceso mediante unha interface de E/S. O programa inclúe unha opción para a execución paso a paso do algoritmo de evolución e un traductor que pode converter os grafcets a tres formatos diferentes: código-P (pseudoensamblador, para ser implementado mediante IL), código C e ecuacións booleanas (para ser implementado mediante LD). Tanto durante a simulación como durante a execución pode verse o estado do programa a través de tres interfaces diferentes:

1. *A interface Grafcet*, na que se anima a evolución da situación do grafcet e os valores das entradas e saídas.
2. *A interface de monitorización de variábeis*, na que se monitorizan os valores de variábeis, entradas e saídas, e os estados de etapas e transicións.
3. *A interface de operador*, na que pode verse unha animación gráfica do proceso. O usuario pode modificar esta interface implementando un conxunto de funcións reservadas no Graf7-C para este cometido.

As variábeis e parámetros precisos para conectar o PC ao proceso mediante unha interface de E/S son definidos nun arquivo denominado *grafcet.io*. Por defecto o Graf7-C declara trinta e dúas entradas (*a-z*) e dezaseis saídas (SB_0-SB_{15}) booleanas, catro entradas (EA_0-EA_3) e dúas saídas (SA_0-SA_1) analóxicas e oito contadores (C_1-C_8), e presupón a existencia de dúas tarxetas de E/S, unha dixital e unha analóxica, cos seus portos de E/S mapeados en memoria. Ademais,

e para permitir a simulación, tamén asigna as entradas booleanas ao teclado. O usuario pode modificar esta interface editando o arquivo *grafcet.io* e implementando un conxunto de funcións reservadas no Graf7-C para realizar as operacións de E/S.

4.2.9.2.7. O algoritmo de interpretación

O algoritmo de interpretación do Graf7-C é de tipo ARS con detección de ciclos estacionarios. Durante as evolucións internas do modelo os eventos externos foron considerados coma eventos na escala interna e os internos foron tratados internamente (§4.1.2.1, caso 1.a).

As accións, incluídas as impulsionais, son executadas unicamente nas situacións estábeis, e atopáronse problemas durante a comprobación da aplicación das ordes de forzado e bloqueo (§4.1.2.3). Cando o grafcet da Figura 4.3 estaba na situación {1, 4, 8} e se produciu o evento externo $\uparrow a$, a evolución do modelo foi a agardada, executándose a orde de forzado da etapa 5 na situación inestábel (2, 5, 8). Sen embargo, cando se partiu da situación {1, 4, 7} a evolución ante o evento $\uparrow a$ foi {1, 4, 7}→(2, 4, 7)→{2, 5, 7}, da que se deduce que neste caso non se executou a orde de forzado, o cal é incorrecto.

Tamén se atoparon problemas na aplicación xerárquica das ordes de forzado. Partindo da situación {1, 10, 20, 30} no grafcet da Figura 3.37, cando a variábel *a* se activou, o modelo evoluiu á situación {2, 10, 21, 32} dacordo ao comportamento agardado. Sen embargo, cando se produce o evento $\uparrow b$, a situación pasou a ser a {3, 10, 21, 32}, que non coincide con ningunha das catro previstas. En efecto, ao desactivarse a etapa 2 e activarse a 3, a orde de forzado sobre a etapa 21 deixa de estar activa, co que a situación do grafcet parcial GP3 podería evoluir á etapa 22, cousa que non acontece. O mesmo pode dicirse da orde de forzado da etapa 21 e o grafcet parcial GP4.

No referente á resposta do Grafc7-C cando varias accións modifican simultaneamente unha mesma variábel (§4.1.2.2), os resultados nos diferentes casos probados foron os seguintes: no caso (1.a) a saída mantivo o valor 1, o que indica que a acción R non é prioritaria sobre as demais en contra do definido no estándar IEC 61131-3; o caso (1.b) non foi considerado coma un erro e a saída tomou o valor resultado da disxunción lóxica das accións activas; e, por último, os casos (1.c) e (1.d) tampouco foron considerados coma erros e a variábel tomou diferentes valores mentres as accións estiveron activas.

4.2.9.3. Conclusións

O Graf7-C é unha aplicación concibida con fines educativos para o desenvolvemento de programas de control mediante Grafcet e C. A aplicación está composta por un editor, un compilador, un traductor (de Grafcet a código-P, C ou ecuacións lóxicas), un simulador, un monitor en tempo real (que permite utilizar o PC como controlador) e unha ampla libraría de funcións C. O usuario pode redefinir partes do programa (a interface de E/S, a interface de operador, etc.) mediante a modificación de funcións C predefinidas que son chamadas durante a execución do programa.

A versión do Grafcet implementada é moi completa, podendo utilizarse todas as estructuras básicas e extensións como as etapas (transicións) fonte e sumidoiro, as macroetapas e as ordes de forzado. As accións e receptividades son codificadas en C7, unha versión da linguaxe C que inclúe algunhas extensións Grafcet como o uso de eventos, temporizadores ou contadores. O algoritmo de interpretación utilizado é de tipo ARS con detección de ciclos estacionarios. No aspecto negativo pode indicarse que o programa non inclúe o concepto de grafcet parcial (as

ordes de forzado aplícanse entre grafcets conexos), permite a edición de macroetapas e macroexpansións incorrectas, deixa baixo a responsabilidade do usuario a utilización correcta das ordes de forzado e non manexa correctamente os conflictos entre accións. En resume, a pesar de ser un programa para o sistema operativo DOS que traballa en modo texto, o Graf7-C implementa case todas as características do Grafcet, proporciona unha linguaxe de alto nivel para a especificación de accións e receptividades, inclúe boas opcións de simulación e monitorización e facilita os medios para conectar o PC ao proceso a través de diferentes interfaces de E/S.

4.3. Conclusións

Os datos obtidos da análise de características de cada aplicación resúmense nas táboas situadas ao final deste capítulo. En base a eses datos poden extraerse as conclusións seguintes:

- No referente ao editor Grafcet:
  1. Todas as aplicacións analizadas (excepto GrafcetView) utilizan un editor que divide o área de traballo en filas e columnas e restrinxe o tipo de nodo que pode inserirse en función da posición. O modelado de estructuras Grafcet complexas require que esta división non impida o uso flexíbel das estructuras de inicio e fin de selección (paralelismo), que se permitan os cruces de liñas ou a utilización de referencias de salto, e que non se limite o número de grafcets conexos nin de etapas iniciais. Isto non é soportado por todas as aplicacións analizadas, en especial as baseadas no SFC.
  2. A edición da estructura xerárquica dun modelo é mais fácil se se dispón dunha vista tipo árbore na que se representen os niveis de aniñamento da xerarquía. Das aplicacións analizadas que soportaban a estructuración dos modelos, a maioría incluían esta vista para a xerárquica estructural mais non para a xerarquía de forzado.
  3. A maior parte das aplicacións fan a análise sintáctica en dúas fases. Isto, xunto coa inclusión de asistentes para a edición de estructuras Grafcet (como o do AutomGen, p.e.) e do código de accións e receptividades (como o do Visual I/O, p.e.), facilita a edición e é especialmente útil para os usuarios inexpertos durante a aprendizaxe.
  4. Ningunha das aplicacións analizadas inclúe un soporte completo á identificación dos diferentes elementos dun grafcet: ou ben non permiten identificar todos os elementos, ou non permiten identificadores alfanuméricos, ou non numeran automaticamente os novos elementos, ou non inclúen opcións que manteñan a coherencia da numeración durante a edición. Neste aspecto, o IsaGraph é a aplicación máis completa, pois permite a numeración alfanumérica de etapas, transicións e subgrafcets, e actualiza correctamente as copias de elementos xa existentes, as referencias de salto e o código de accións e receptividades durante a edición.
  5. Outra característica desexábel nun editor Grafcet é a posibilidade de xestionar unha libraría de estructuras grafcet predefinidas. Unicamente o AutomGen inclúe unha destas librarías, mais non permite modificala.

- No referente ás opcións de simulación e execución:
  1. O Grafcet é un formalismo definido para ser independente da tecnoloxía utilizada na súa implementación. Para incrementar a súa aplicabilidade é preciso dispor dun ambiente de execución que sexa facilmente portábel e dende o que poidan utilizarse múltiples sistemas E/S e protocolos de intercambio de información con outras aplicacións. Entre as aplicacións analizadas a opción máis común é a de dispor dun

emulador PLC (módulo softPLC) no que poidan cargarse os modelos previamente compilados ou traducidos a unha linguaxe intermedia que é interpretada polo emulador. Algunhas aplicacións inclúen tamén unha libraría de programación que permite engadir novos "drivers" de E/S e funcións adicionais ao emulador. O ambiente de execución máis completo dos analizados é o do IsaGraph, que está dispoñíbel para múltiples sistemas operativos de tempo real e que inclúe unha libraría para desenvolver "drivers" de E/S e portar o ambiente a novos sistemas.

2. A simulación dos modelos require soporte á animación gráfica da súa evolución, monitorización en tempo real dos valores das variábeis e algún método de simulación das entradas. Todas as aplicacións analizadas, excepto o PL7, inclúen estas opcións.
3. O soporte para a depuración dos modelos require opcións que permitan a execución ciclo a ciclo, colocación de puntos de parada baseados na situación actual do modelo, forzado da situación, e modificación e forzado de variábeis. Unicamente o IsaGraph e o PL7 inclúen todas estas opcións, nas demais aplicacións o mais común é permitir a monitorización de variábeis e nalgunhas delas a execución ciclo a ciclo.

- No referente á sintaxe do Grafcet:
  1. O soporte sintáctico ofrecido polas aplicacións analizadas varía dependendo de si a aplicación está orientada á edición de SFCs ou á edición de versións mais completas do Grafcet. Nas aplicacións do primeiro tipo non poden utilizarse etapas (transicións) fonte e sumidoiro, non permiten o uso flexíbel das estructuras de inicio e fin de selección (paralelismo), e non permiten representar semáforos. Ademais algunhas destas aplicacións unicamente permiten editar unha rede SFC e limitan o número de etapas iniciais. Entre as do segundo tipo, o WinGrafcet presenta as mesmas limitacións que as aplicacións SFC, o PL7 non implementa as etapas (transicións) fonte e sumidoiro, e as referencias de salto non son incluídas en todas as aplicacións.
  2. A implementación das xerarquías do Grafcet é un dos aspectos nos que se detectaron maiores diferencias e deficiencias no que á sintaxe respecta. As aplicacións SFC non implementan nin a xerarquía estructural nin a de forzado, mentres que das demais só algunhas implementan as macroetapas e unicamente o AutomGen e o Graf7-C soportan as ordes de forzado. Ningunha das aplicacións analizadas implementa de forma explícita a estructuración dos modelos en grafcets globais, parciais e conexos, nin permite visualizar ou comprobar a coherencia e a corrección da xerarquía de forzado.
  3. Outro aspecto sintáctico no que hai grandes diferencias é a implementación de accións e receptividades. Hai aplicacións que implementan completa ou parcialmente o estándar IEC (MachineShop, Actwin), outras que implementan mediante unha sintaxe propia parte dos tipos de accións definidos polo IEC (AutomGen, GrafcetView) e outras que utilizan unha linguaxe de programación de alto nivel e inclúen medios para representar diferentes comportamentos temporais nas accións (Visual I/O, Graf7-C). A maioría das aplicacións (excepto o Visual I/O) permiten utilizar eventos, temporizadores e contadores nas accións e receptividades, e só algunhas inclúen algún medio para impedir os 'efectos colaterais' durante a avaliación das receptividades.
  4. Todas as aplicacións detectan durante a verificación sintáctica (xa sexa nunha ou en dúas fases) os erros de edición mais comúns: alternancia etapa-transición, erros sintácticos en accións e receptividades ou elementos sen identificador. A maioría tamén detecta ou impide os identificadores duplicados e as referencias a etapas inexistentes. Sen embargo características mais avanzadas como a utilización dunha

transición simultaneamente como fonte e sumidoiro ou a estructura sintáctica das macroexpansións non se comproba case en ningunha aplicación, e tampouco hai ningunha que comprobe a corrección e coherencia das ordes de forzado.

- No referente á interpretación dos modelos:
  1. O algoritmo de interpretación máis utilizado é o SRS, utilizando en todas as aplicacións IEC e as orientadas á programación de PLCs. Unicamente tres aplicacións implementan un algoritmo tipo ARS, e destas unicamente dúas (GrafcetView, Graf7-C) detectan os ciclos estacionarios durante a execución. Ningunha permite configurar como serán considerados os eventos ou que valor das variábeis (interno ou externo) será utilizado durante as evolucións internas, e tampouco permiten indicar que accións se executarán unicamente nas situacións estábeis e cales en calquera situación.
  2. Só as aplicacións AutomGen e Graf7-C implementan as ordes de forzado e ningunha delas as aplica correctamente durante a execución: no Graf7-C non teñen prioridade sobre as evolucións estructurais e en ningunha das dúas se aplican recursivamente dacordo aos niveis da xerarquía de forzado.
  3. Das tres aplicacións que implementan un algoritmo de interpretación tipo ARS, ningunha aplica correctamente as accións e ordes de forzado durante as evolucións internas do modelo: o GrafcetView non ten ordes de forzado e só implementa accións externas que se executan nas situacións estábeis; o AutomGen executa accións e ordes de forzado en todas as situacións, tanto estábeis como inestábeis; e o Graf7-C executa as accións só en situacións estábeis, independentemente do seu tipo, e aplica de forma incorrecta as ordes de forzado.
  4. Ningunha das aplicacións analizadas detecta ou impide os conflictos entre accións. Algunhas das aplicacións que seguen as recomendacións do IEC (ActWin, IsaGraph) dan prioridade ás asociacións tipo R e a maioría aplican a disxunción lóxica dos resultados en caso de conflicto. Sen embargo ningunha implementa a semántica definida polo IEC mediante os bloques de control de accións (§3.6.1.3) e tampouco impiden a aparición de comportamentos indeterministas cando hai asignacións simultáneas de valores múltiples a unha mesma variábel.
  5. Das aplicacións que permiten a utilización de transicións fonte (catro en total) só o AutomGen e o Graf7-C as consideran como sempre validadas durante a execución.

- Outras consideracións:
  1. Exceptuando o GrafcetView, que implementa o Grafcet como libraría para LabView, ningunha das aplicacións analizadas permite utilizar o Grafcet fora do ambiente da propia aplicación, de xeito que poida ser integrado con outras aplicacións, librarías ou linguaxes. Isto ten como consecuencia a perda de aplicabilidade, xa que non permite utilizar o Grafcet como formalismo xenérico independente da tecnoloxía de implementación e aberto á integración de calquera linguaxe para a especificación de accións e receptividades.
  2. Non existe un formato para o intercambio de grafcets entre aplicacións. Isto é consecuencia tanto do comentado no punto anterior como das diferencias entre as aplicacións á hora de implementar o Grafcet. Este formato nin sequera existe entre as aplicacións que seguen as recomendacións do IEC.
  3. Ningunha aplicación permite configurar o tipo de interpretación co que o usuario quere executar os modelos. Sería desexábel dispor dun ambiente de execución no que o

usuario puidera elixir entre executar un modelo mediante unha interpretación SRS ou ARS sen necesidade de modificar o modelo.

4. As aplicacións con algoritmos ARS tampouco permiten modificar a configuración de como se considerarán os eventos, variábeis e accións nas dúas escalas de tempo deste algoritmo. Ao igual que no punto anterior, sería desexábel que o usuario puidera configurar estas opcións durante a execución sen que iso afectase ao modelo.
5. O soporte á distribución dos modelos Grafcet é limitado. A maioría das aplicacións analizadas non permiten distribuílos entre varios ambientes de execución, e nas que o permiten (IsaGraph, p.e.) a distribución é ríxida: primeiro defínense os nodos do sistema e, posteriormente, edítanse os grafcets de cada nodo. Sería desexábel dispor dun editor que permitira editar os modelos Grafcet de maneira independente da estructura do sistema de control, que incluíra ferramentas que permitiran analizar diferentes posibilidades de distribución e que realizaran esta de forma automática.

	GRAFCETVIEW	ISAGRAPH	MACHINESHOP	PL7
Datos da aplicación				
Dispoñibilidade	Comercial	Comercial	Comercial	Comercial
Fabricante	TecAtlant	AlterSys	CTC Parker Automation	Schneider Automation
Tipo	Librarfa LabView	Programación sistemas distribuídos	HMI/SCADA	Programación PLC
Sistema operativo	Windows, Unix	Windows	Windows	Windows
Sistema E/S	Serie, DAQ, GPIB, VXI	Modbus, Memobus, Profibus	Profibus, DeviceNet, Modbus, PC104	Modbus, Profibus, InterBus, CanOpen
Equipo de control	PC	PC, PLC, SCADA	PC, PLC, CTC	PLCs Telemecanique
Ambiente de execución	LabView	Linux, VxWorks, pSOS, Windows	RTXDOS	PLCs Telemecanique
Editor Grafcet				
Tipo editor	Editor LabView	Filas e columnas	Filas e columnas	Filas e columnas
Edición: copiar, mover, ...	Si	Si	Si	Si
Análise sintáctica	Si (2 fases)	Si (2 fases)	Si (2 fases)	Si (2 fases)
Asistente edición código	Non	Si	Si	Non
Asistente edición estructuras	Non	Non	Non	Non
Librarfa de estructuras	Non (pode crearse)	Non	Non	Non
Identificación automática	Non	Si	Si	Si
Opcións renumeración	Non	Si	Non	Non
Renumeración ao copiar	—	Si	—	—
Renum. referencias de salto	—	Si	—	—
Renumeración referencias en accións e receptividades	—	Si	—	—
Opcións de simulación e depuración				
Animación de grafcets	Non (pode emularse)	Si	Si	Si
Simulación	Si (mediante LabView)	Si	Si	Non
Ambiente de simulación	LabView	SoftPLC portábel	SoftPLC en RTXDOS	—
Simulación de entradas	Si (mediante LabView)	Si	Si	—
Depuración	Si (mediante LabView)	Si	Si	Si
Execución paso a paso	Si (mediante LabView)	Si	Non	Si
"Breakpoints"	Si (mediante LabView)	Si	Non	Si
Monitorización variábeis	Si (mediante LabView)	Si	Si	Si
Forzado de variábeis	Si (mediante LabView)	Si	Si	Si
Funcións adicionais				
Estructuración do proxecto	Si (capsulamento VIs)	Si (equivalente a IEC)	Si (IEC)	Si
Diccionario / Táboa símbolos	Non	Si	Si	Si
Editor configuración hardware	Si (mediante LabView)	Si	Si	Si
Editor configuración E/S	Si (mediante LabView)	Si	Si	Si
Referencias cruzadas	Non	Si	Si	Si
Biblioteca de programas	Si (librarías de VIs)	Si	Si	Si
Editor interface de usuario	Si (mediante LabView)	Non	Si (mediante Interact)	Si
Documentación do proxecto	Limitada (docum. VIs)	Si	Si	Si
SDK para programación	Non	Si	Si	Si

Táboa 4-VII. Resume dos resultados da análise: funcionalidades das aplicacións.

VISUAL IO	AUTOMGEN	ACTWIN	WINGRAFCET	GRAF7-C
Datos da Aplicación				
Comercial	Comercial	Comercial	Comercial/Educación	Educación
ArSoft International	Irai	Actron	CDRP Montpellier	CCDMD
HMI/SCADA	Programación software de control	Programación PLC	Programación Grafcet	Programación Grafcet
Windows	Windows	Windows	Windows	DOS
Profibus, Modbus, Interbus, CanOpen	Profibus, Modulink, Modbus	Profibus, Interbus, CANOpen, DeviceNet	AM1, AM2, IP16, Approtech, Ipocaen	Tarxetas E/S para PC
PC	PC, PLC	PC, PLCs Hitachi	PC	PC
Windows/Sistemas RT x86	Postprocesador PC ou PLC	Windows	Windows	DOS
Editor Grafcet				
Filas e columnas	Filas e columnas	Edición controlada	Filas e columnas	Filas e columnas
Si	Si	Si	Si	Si
Si (2 fases)	Si (1 fase)	Si (2 fases)	Si (1 fase)	Si (2 fases)
Si	Si	Si	Non	Non
Non	Si	Non	Non	Non
Non	Si	Non	Non	Non
Si	Non	Si	Si	Si
Si	Si	Si	Si	Si
Non	Non	—	Non	Si
Si	—	Si	Non	Si (con erros)
Non	Non	Si	Non	Non
Opcións de simulación/depuración				
Si	Si	Si	Si	Si
Si	Si	Si	Si	Si
SoftPLC en Windows	Simulador Windows	SoftPLC en Windows	Simulador Windows	Simulador DOS
Si	Si	Si	Si	Si
Si	Si	Si	Si	Si
Non	Si	Non	Non	Si
Non	Non	Non	Non	Non
Si	Si	Si	Si	Si
Si	Si	Si	Non	Non
Funcións adicionais				
Si	Si	Si (IEC)	Non	Non
Si	Si	Si	Non	Non
Si	Si	Si	Non	Non
Si	Si	Si	N/A	Si (mediante prog.)
Si	Non	Si	Non	Non
Si (mediante VPUs)	Si	Si	Non	Si
Si	Si	Non	Non	Si (mediante prog.)
Si	Si	Si	Non	Non
Si	Non	Si	Non	Si

Táboa 4-VIII. Resume dos resultados da análise: funcionalidades das aplicacións (continuación).

	GRAFCETVIEW	ISAGRAPH	MACHINESHOP	PL7
Grafcet / SFC	Grafcet	SFC+xerarquía propia	SFC	Grafcet
Sintaxe				
Elementos básicos				
Etapa/Etapa Inicial/Transición	Si	Si	Si	Si
Múltiples etapas iniciais	Si	Si	Non	Si
Receptividades/Accións	Si	Si	Si	Si
Referencias de salto	Non	Si	Si (etapas de salto)	Si
Etapa fonte/sumidoiro	Si	Non	Etapas sumidoiro	Non
Transición fonte/sumidoiro	Si	Non	Non	Non
Estructuras de control				
Secuencia	Si	Si	Si	Si
Selección de secuencia	Si (uso flexíbel)	Si (uso flexíbel)	Si (uso ríxido)	Si (uso flexíbel)
Paralelismo	Si (uso flexíbel)	Si (uso flexíbel)	Si (uso ríxido)	Si (uso flexíbel)
Salto de secuencia	Si	Si	Si	Si
Ciclo	Si	Si	Si	Si
Semáforo	Si	Non	Non	Si
Estructura xerárquica				
Macroetapas	Si (capsulamento VIs)	Non	Non	Si
Grafcet Conexo/Parcial/Global	Si (capsulamento VIs)	Non	Non	Non
Múltiples grafcets conexos	Si	Si	Non	Si
Ordes de forzado	Non	Non	Non	Non
Accións / Receptividades				
Linguaxes	Sintaxe propia	IL, ST, LD	IL, ST, FBD, LD	IL, LD, FBD, ST
Tipos de accións	N, C emulación D e L	N, R, S, P0, P1	N, R, S, P, L, D, DS, SD, SL	P0, N, P1
Variábeis de etapa	Si	Si	Si	Si (uso limitado)
Eventos	Si	Si (FBs IEC)	Si (FBs IEC)	Non
Temporizadores	Si (variábeis de etapa)	Si (FBs IEC)	Si (FBs IEC)	Si (FBs IEC)
Contadores	Non	Si (FBs IEC)	Si (FBs IEC)	Si (FBs IEC)
Efectos colaterais	Si	Si	Si	Non
Identificación				
Tipo	Numérico	Alfanumérica	Alfanumérica	Numérico
Elementos	Etapas	Etapas, trans., grafcets	Etapas, trans., accións	Etapas
Erros detectados na análise sintáctica				
Alternancia Etapa/Transición	Si	Si	Si	Si
Identificadores duplicados	Si	Si	Si	Si
Código accións	Si	Si	Si	Si
Código receptividades	Si	Si	Si	Si
Ref. de salto inexistentes	—	Si (evítase na edición)	Si	Si
Transición fonte e sumidoiro	Si	—	—	—
Estructura macroexpansións	Non	—	—	Si
Coherencia xerarquía forzado	—	—	—	—
Interpretación				
Algoritmo				
Tipo	ARS	SRS	SRS	SRS
Detección ciclos estacionarios	Si	—	—	—
Trans. fonte sempre validadas	Non	—	—	—
Eventos na escala interna	Eventos	—	—	—
Escala eventos internos	Interna	—	—	—
Variábeis externas/internas	Non	—	—	—
Accións en situacións inestábeis	Non	—	—	—
Accións externas/internas	Non	—	—	—
FO en situacións inestábeis	—	—	—	—
Aplicación recursiva de FOs	—	—	—	—
Conflictos entre accións				
Prioridade do Reset	—	Si	Si	N/A
Temporización var. booleanas	Non	—	Non	N/A
Temporización de asignacións	—	—	Non	N/A
Asignacións múltiples	—	Non	Non	N/A

Táboa 4-IX. Resume dos resultados da análise:características Grafcet soportadas.

VISUAL I/O	AUTOMGEN	ACTWIN	WINGRAFCET	GRAF7-C
Grafcet	Grafcet	SFC	Grafcet	Grafcet
Sintaxe				
Elementos básicos				
Si	Si	Si	Si	Si
Si	Si	Non	Si	Si
Si	Si	Si	Si	Si
Si	Non	Si	Si	Si
Si	Si	Non	Non	Si
Si	Si	Non	Non	Si
Estructuras de control				
Si	Si	Si	Si	Si
Si (uso flexíbel)	Si (uso flexíbel)	Si (uso ríxido)	Si (uso ríxido)	Si (uso flexíbel)
Si (uso flexíbel)	Si (uso flexíbel)	Si (uso ríxido)	Si (uso ríxido)	Si (uso flexíbel)
Si	Si	Si	Si	Si
Si	Si	Si	Si	Si
Si	Si	Non	Non	Si
Estructura xerárquica				
Non	Si	Non	Non	Si (tamén tarefas)
Non	Non (agrupam. GCs)	Non	Non	Non
Si	Si	Si	Si	Si
Non	Si	Non	Non	Si
Accións / Receptividades				
Pascal	Sintaxe propia	LD, FBD	Sintaxe propia	C7
N emulación L, D, P	N, C, R, S, P1, P0 emulación L, D, DS	N, R, S, P, L, D, DS, SD, SL	N	N, C, R, S, P emulación L, D
Si	Si	Si	Si	Si
Non	Si	Non	Si	Si
Non	Si	Si	Si	Si
Non	Si	Si	Si	Si
Si	Non	Si	Non	Si
Identificación				
Numérico	Numérico	Alfanumérico	Numérico	Num./Alfanumérico
Etapas	Etapas	Etapas, accións, recs.	Etapas	Etapas, trans., GCs
Erros detectados na análise sintáctica				
Si (permite erros)	Si	Si	Si	Si
Non	Si	Si	Non	Si
Si	Si	Si	Si	Si
Si	Si	Si	Si	Si
Si (evítase na edición)	—	Si (evítase na edición)	Si	Non
Non	Non	—	—	Non
—	Si (limitado)	—	—	Non
—	Non	—	—	Non
Interpretación				
Algoritmo				
SRS	ARS	SRS	SRS	ARS
—	Non	—	—	Si
Non	Si	—	—	Si
—	Eventos	—	—	Eventos
—	Interna	—	—	Interna
—	Non	—	—	Non
—	Si	—	—	Non (nin impulsionais)
—	Non	—	—	Non
—	Si	—	—	Non
—	Non	—	—	Non
Conflictos entre accións				
—	Non	Si	N/A	Non
Non	Non	Non	N/A	Non
Non	Non	Non	N/A	Non
Non	Non	Non	N/A	Non

Táboa 4-X. Resume dos resultados da análise:características Grafcet soportadas (continuación).

Capítulo 5. Proposta dun metamodelo para o Grafcet

Como se indicou no Capítulo 4, dende a aparición do primeiro estándar Grafcet internacional [84], fixéronse diferentes propostas coa intención de mellorar as capacidades de do Grafcet para o modelado de sistemas complexos. Algunhas destas propostas foron recollidas na norma [2] e, recentemente, nunha nova revisión do estándar internacional [85]. Ademais tamén se estandarizou unha versión orientada á programación de PLCs [86]. A existencia de diferentes versións do estándar e distintas propostas de mellora do formalismo levou na práctica a unha situación, posta de relevo nas conclusións da análise realizada no Capítulo 4, na que cada aplicación implementa versións diferentes do Grafcet dando lugar á existencia de múltiples dialectos. Algunhas das desvantaxes desta situación son:

1. Non se dispón dunha referencia común que permita verificar que unha aplicación implementa as características definidas polos estándares.
2. Non existe un formato estándar de intercambio de grafcets entre aplicacións.
3. Non se dispón da infraestructura precisa (p.e. libraría, compilador, etc.) que permita integrar o Grafcet como linguaxe xenérica de especificación de secuencias de control en calquera aplicación.

Coa intención de solucionar estes problemas e formalizar as características do Grafcet utilizadas na ferramenta proposta nesta tese de doutoramento, proponse neste capítulo un metamodelo que describe a sintaxe do Grafcet[37] [132], dacordo ás definicións e propostas anteriores á última revisión do estándar.

Un metamodelo é '*unha definición precisa dos conceptos, regras e restriccións utilizados na creación de modelos*'. É dicir, o metamodelo proposto formaliza os conceptos e regras utilizados durante a especificación da secuencia de control dun sistema modelado mediante Grafcet. Algunha das avantaxes de dispor dun metamodelo son as seguintes:

1. Pode ser utilizado como base para a definición dun formato de almacenamento e intercambio de información entre aplicacións.
2. Existen ferramentas baseadas no uso de metamodelos para a análise formal, verificación sintáctica e traducción de modelos.
3. É un elemento de integración cos metamodelos doutros formalismos, incrementándose así a reusabilidade e aplicabilidade dos modelos.

[37] Os aspectos semánticos son tratados en (§8.4).

O resto do capítulo estructurase do xeito seguinte: no apartado (§5.1) descríbese o metamodelo proposto; no apartado (§5.2) preséntanse os aspectos máis relevantes dunha libraría C++ que implementa o metamodelo; en (§5.3) móstranse algúns exemplos de modelado utilizando esta libraría e, por último, no apartado (§5.4) resúmense as conclusións do capítulo.

5.1. Metamodelo para a sintaxe do Grafcet

O metamodelo que se propón toma como base o definido en [59]. Mellorouse a representación das xerarquías Grafcet, engadiuse unha descrición precisa das accións e tipos de datos utilizados nos modelos e definiuse un conxunto de invariantes que describen as regras que debe cumprir un modelo para ser sintacticamente correcto. Ademais definíronse as relacións entre as metaclases do Grafcet e as do UML, de xeito que poida ser utilizado como alternativa aos StateCharts na especificación do comportamento das compoñentes dinámicas dun modelo. A representación do metamodelo fíxose utilizando os mesmos medios que utiliza UML: a MOF [127] e a linguaxe OCL [178].

5.1.1. Estructura de paquetes

O metamodelo está estructurado en catro paquetes (Figura 5.1):

1. *Grafcet data*, no que se definen os tipos de datos, variábeis e expresións do metamodelo.
2. *Grafcet core*, no que se especifican os elementos sintácticos básicos do Grafcet.
3. *Grafcet actions*, no que se inclúen as definicións dos tipos de accións.
4. *Grafcet hierarchy*, no que se definen as metaclases que permiten estructurar xerarquicamente os modelos.

Como todos os paquetes dependen do metamodelo UML (Figura 5.2), antes de pasar a describir o metamodelo Grafcet en detalle, reprodúcense aquí parte das especificacións das metaclases UML[38] utilizadas, para por en contexto o metamodelo proposto.

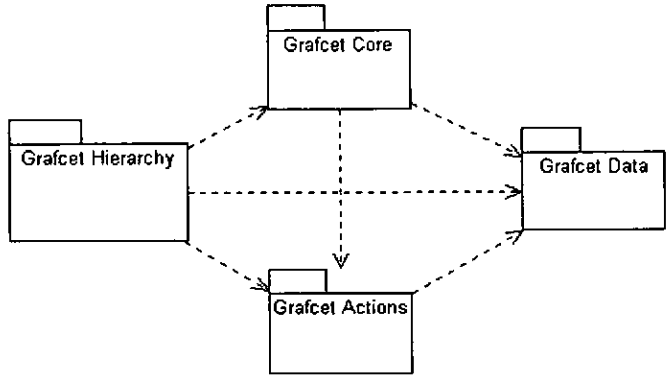


Figura 5.1. Estructura de paquetes do metamodelo Grafcet.

5.1.1.1. Metaclases do metamodelo UML

Na Figura 5.3 pode verse un diagrama de clases contendo todas as metaclases UML utilizadas directa ou indirectamente polo metamodelo Grafcet proposto. A descrición resumida de cada metaclase é a seguinte:

[38] Pode consultarse en [127] a especificación completa do metamodelo.

Elemento del modelo ("Model Element")

É un elemento que representa unha abstracción extraída do sistema modelado. No metamodelo é unha entidade con identidade nun modelo. É a metaclase base do metamodelo UML, todas as demais metaclases derivan directa ou indirectamente dela.

Atributos:

- *name*, identificador do elemento.

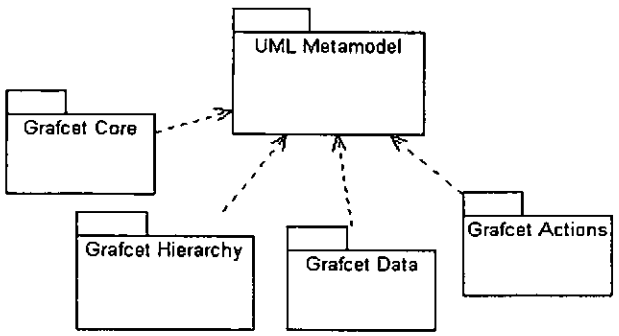


Figura 5.2. Dependencias do metamodelo UML.

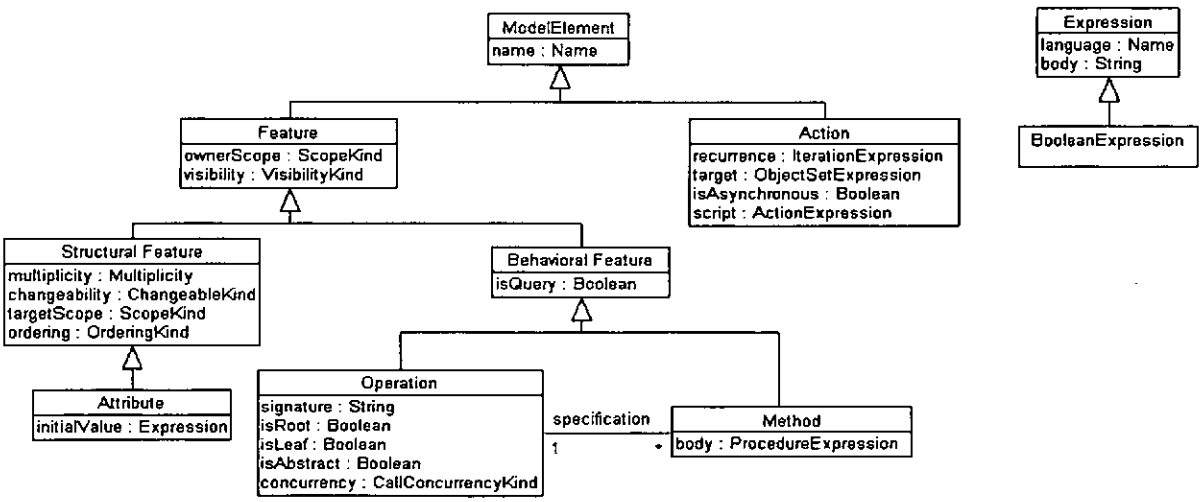


Figura 5.3. Diagrama parcial das clases do metamodelo UML.

Característica ("Feature")

É unha propiedade capsulada nun clasificador (elemento con propiedades estáticas e dinámicas). No metamodelo declara unha característica estructural ou de comportamento dun clasificador ou dunha instancia dun clasificador. Esta metaclase é abstracta.

Atributos:

- *ownerScope*, indica se a característica aparece en cada instancia do clasificador ou é única. Pode ter un dos valores seguintes:
  - *instance*, cada instancia ten o seu propio valor para a característica.
  - *classifier*, hai un único valor da característica para todas as instancias do clasificador.
- *visibility*, indica se a característica pode ser utilizada por outros clasificadores. Pode ter un dos valores seguintes:

- *public,* calquera clasificador externo pode usar a característica.
- *protected,* calquera descendente do clasificador pode usar a característica.
- *private,* unicamente o clasificador pode usar a característica.
- *package,* calquera clasificador no mesmo paquete declarado como propietario da característica pode utilizala.

Característica estructural ("Structural Feature")

Refírese a unha característica estática dun elemento, como por exemplo un atributo. No metamodelo, unha característica estructural declara un aspecto estructural dunha instancia dun clasificador. A característica pode especificar, por exemplo, a multiplicidade dun atributo. Esta é unha metaclase abstracta.

Atributos:

- *changeability*, indica se pode modificarse o valor despois da creación do obxecto. Os valores que pode tomar son:
  - *changeable*, pode modificarse.
  - *frozen*, o valor non pode ser modificado.
  - *addOnly*, utilizado cando a multiplicidade é variábel. Poden engadirse novos valores, mais unha vez creados non poden modificarse nin eliminarse.
- *multiplicity*, indica o número de valores da característica que pode haber en cada instancia. O caso mais común é unha multiplicidade 1..1 (un único valor para a característica).
- *ordering*, indica se o conxunto de instancias está ordenado. A ordenación é mantida polas operacións que engaden valores á característica. Este atributo ten significado só cando a multiplicidade é superior a un. Os valores que pode ter son:
  - *unordered*, os valores non están ordenados.
  - *ordered*, un conxunto de instancias pode ser percorrido seguindo unha orde.
  - outros valores (p.e. *clasificado*) definidos polo usuario.
- *targetScope*, indica se os valores da característica se refiren a instancias ou a clasificadores. Pode ter un dos seguintes valores:
  - *instance*, cada valor é unha referencia a unha instancia do clasificador. Este é o valor normal dun atributo.
  - *classifier*, cada valor contén unha referencia ao clasificador. Este valor sirve para almacenar metainformación.

Atributo ("Attribute")

Un atributo é un espacio identificado dentro dun clasificador que describe o rango de valores que as súas instancias poden tomar.

Atributos:

- *initialValue*, unha expresión que especifica o valor do atributo despois da iniciación. Este atributo está pensado para ser avaliado no momento en que o obxecto é iniciado.

Característica dinámica ("Behavioral Feature")

Refírese a unha característica dinámica dun elemento, como por exemplo unha operación ou método. No metamodelo especifica un aspecto do comportamento dun clasificador. Todos os aspectos dinámicos dun clasificador, como as operacións e métodos, son subclases desta metaclase que é abstracta.

Atributos:

- *isQuery*, indica si a execución da característica modifica o estado do sistema. O valor *true* indica que non o modifica e *false* que si.

Operación ("Operation")

Unha operación é un servicio que pode solicitarse dende un obxecto para levar a cabo un comportamento. Unha operación ten unha declaración que describe os seus parámetros e valores devoltos. No metamodelo unha operación é unha característica dinámica que pode aplicarse ás instancias do clasificador que contén a operación.

Atributos:

- *signature*, declaración da operación.
- *concurrency*, especifica a semántica cando hai varias chamadas concorrentes á mesma instancia pasiva. Os valores posíbeis son:
  - *sequential*, os clientes deben coordinarse para que só unha chamada estea activa en cada momento. A semántica e integridade do sistema non pode garantirse se hai varias chamadas simultáneas.
  - *guarded*, están permitidas varias chamadas simultáneas mais só pode comezar unha delas. As demais son bloqueadas ate que a primeira se completa. É responsabilidade do deseñador do sistema garantir que non se produzan interbloqueos ("deadlocks").
  - *concurrent*, pode haber múltiples chamadas simultáneas dende "threads" concorrentes e todas elas son executadas concorrentemente mantendo a corrección semántica.
- *isAbstract*, indica se a operación ten implementación. Se é *true* a implementación é proporcionada por unha clase derivada, se é *false* impleméntase na propia clase ou hérdase.
- *isLeaf*, indica se a operación pode ser redefinida nunha clase derivada.
- *isRoot*, indica se a clase pode herdar ou non unha declaración para a mesma operación. O valor *true* indica que non e *false* que si.

Método ("Method")

Un método é a implementación dunha operación. Especifica o algoritmo ou procedemento que calcula os resultados da operación. No metamodelo un método é unha declaración dunha parte identificada do comportamento dun clasificador e realiza unha (directamente) ou varias (indirectamente) operacións do clasificador. Como moito debe haber un único método para cada par clasificador/operación (o primeiro como propietario e a segunda como especificación do método).

Atributos:

- *body*, implementación do método.

Asociacións:

- *specification*, indica a operación que o método implementa.

Acción ("Action")

Unha acción é a especificación dunha sentencia executábel, abstracción dun procedemento computacional que provoca un cambio no estado do modelo e que pode realizarse mediante o envío dunha mensaxe a un obxecto ou a modificación dunha conexión ou valor dun atributo. Esta metaclase é abstracta.

Atributos:

- *isAsynchronous*, indica si o envío dun estímulo por parte da acción é asíncrono ou non.
- *recurrence*, expresión que determina cantas veces debe realizarse a acción.
- *script*, expresión que describe os efectos da acción.
- *target*, expresión que indica o conxunto de instancias aos que se lles vai a aplicar a acción. UML non define se a acción é aplicada secuencialmente ou en paralelo.

Expresión ("Expression")

Unha expresión define unha sentencia que devolverá un conxunto de instancias cando sexa avaliada nun contexto. As expresións non modifican o contorno no que son avaliadas.

Atributos:

- *language*, indica con que linguaxe se especifica a expresión. As predefinidas son:
  - *OCL*, a linguaxe utilizada en UML [178] para a expresión de restricións.
  - *A linguaxe natural*, indicada cun valor baleiro no atributo *language*.
- *body*, contén o texto da expresión na linguaxe indicada.

Expresión booleana ("Boolean Expression")

Especifica unha sentencia que devolverá un valor booleano ao ser avaliada.

5.1.2. Sintaxe abstracta do modelo

Nesta sección especifícase a sintaxe abstracta do metamodelo utilizando os diagramas de clases UML para representar as metaclases e as súas relacións. Para cada metaclase inclúese ademais unha descrición informal da súa semántica e descríbese o significado dos seus atributos e asociacións.

5.1.2.1. O paquete *Grafcet Data*

No paquete *Grafcet Data* defínense os tipos de datos e as metaclases que permiten modelar a información e as expresións utilizadas nos modelos Grafcet. A Figura 5.4 mostra o diagrama de clases deste paquete no que pode verse a relación entre as metaclases definidas e as metaclases UML comentadas no apartado anterior. A descrición detallada das metaclases deste paquete é a seguinte:

Tipo de etapa ("StepType")

Enumeración que define o tipo dunha etapa do modelo. Os valores escalares definidos son:

- *step*, indica que a etapa é unha etapa simple.
- *initial*, a etapa é unha etapa inicial.
- *macro*, a etapa representa unha macroetapa, cuxos contidos serán especificados nunha macroexpansión.

Tipo de nodo ("NodeType")

Enumeración que define o tipo dun nodo do modelo dende o punto de vista da súa posición nunha secuencia de control. Os valores escalares definidos son:

- *source*, o nodo unicamente pode aparecer ao inicio dunha secuencia de control (non pode ter conexións con nodos que o precedan na secuencia).

- *target*, o nodo unicamente pode aparecer ao final dunha secuencia de control (non pode ter conexións con nodos que o sucedan na secuencia).
- *both*, o nodo pode aparecer en calquera punto dunha secuencia de control (pode ter conexións con nodos que o precedan ou sucedan na secuencia).
- *none*, o nodo non forma parte dunha secuencia de control (non ten conexións con ningún outro nodo).

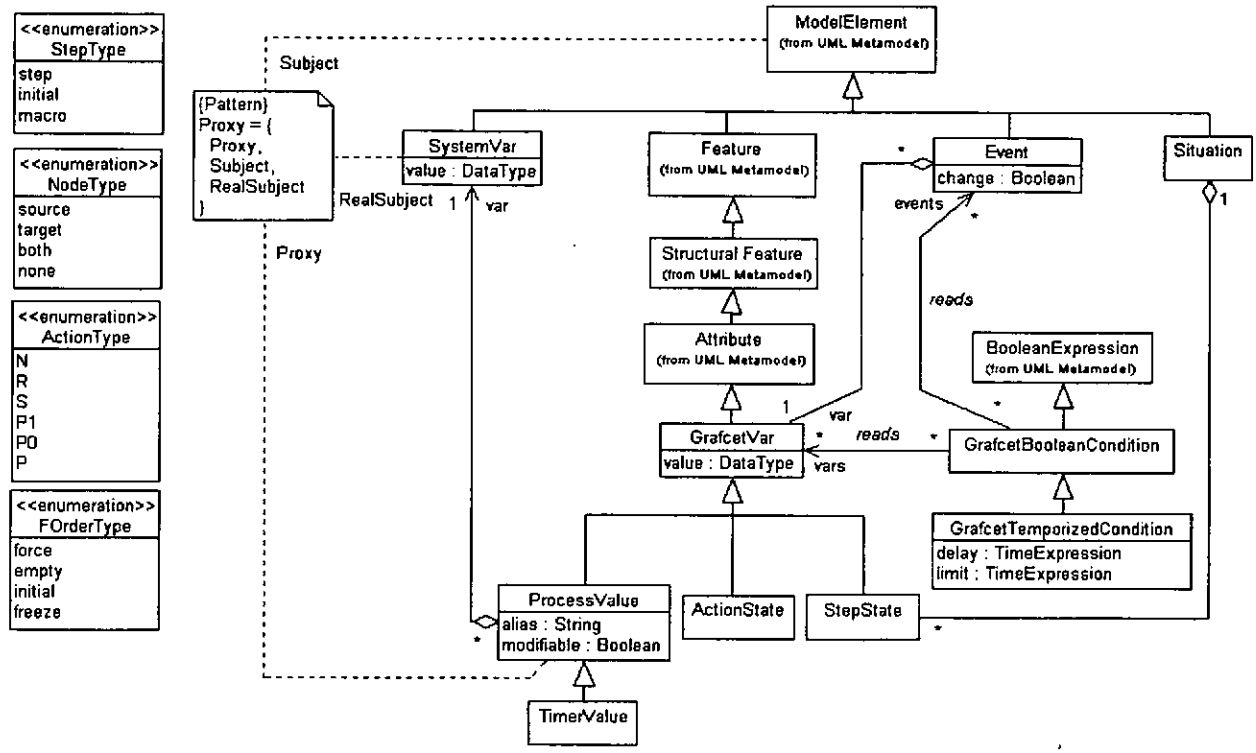


Figura 5.4. Diagrama de clases do paquete *Grafcet Data*.

Tipo de acción ("ActionType")

Enumeración que define a semántica temporal que debe aplicarse durante a activación dun bloque de acción asociado a unha etapa para modificar o valor dunha variábel ou o estado de activación dunha acción. Os valores escalares definidos son: N, R, S, P, P1 e P0.

Tipo de orde de forzado ("ForderType")

Enumeración que define a semántica dunha orde de forzado. Os valores escalares definidos son:

- *force*, a orde forza unha situación específica nun grafcet. As etapas pertencentes á situación dada son activadas e todas as demais desactivadas.
- *empty*, a orde forza a situación baleira nun grafcet. Todas as etapas do grafcet son desactivadas.
- *initial*, a orde forza a situación inicial nun grafcet. Todas as etapas iniciais do grafcet son activadas e todas as demais desactivadas.
- *freeze*, a orde forza a situación actual dun grafcet. Mantense o estado de activación das etapas impedindo as evolucións estructurais do modelo.

Variábel do sistema ("SystemVar")

Representa un valor externo utilizado polo modelo Grafcet. Esta metaclase derivouse de *ModelElement* para permitir que aplicacións específicas do metamodelo a describan con maior detalle. A este nivel de abstracción o método de acceso ou a descrición detallada do elemento non son relevantes, unicamente interesa indicar que é un elemento externo que contén un valor accedido dende o modelo[39].

Atributos:

- *value*, valor do elemento externo accedido dende o modelo.

Variábel do modelo ("GrafcetVar")

Representa as variábeis identificadas no modelo. Mediante o atributo *visibility*, herdado da metaclase *Feature*, pode especificarse a visibilidade dende outros modelos. Poden definirse variábeis accesíbeis dende outros grafcets ou só internamente, xa sexa no propio grafcet ou nos seus derivados. Tamén pode especificarse o nome, modificabilidade, multiplicidade e valor inicial mediante os atributos derivados das clases do metamodelo UML (Figura 5.3).

Atributos:

- *value*, valor da variábel.

Evento ("Event")

Un evento representa un cambio no valor dunha variábel booleana manexada polo modelo. O xeito de xerar o evento ou o seu manexo son aspectos que non se especifican no metamodelo.

Atributos:

- *change*, indica o tipo de cambio do valor: *true* indica un flanco de subida e *false* un de baixada.

Asociacións:

- *var*, variábel cuxo valor cambia.

Valor do proceso ("ProcessValue")

Esta metaclase representa o acceso a unha variábel externa dende o modelo. O medio de acceso non é especificado no metamodelo. Esta metaclase derívase da metaclase *GrafcetVar* para permitir a súa utilización nas condicións e accións do modelo do mesmo xeito que calquera outra variábel.

Atributos:

- *alias*, nome utilizado no modelo para referirse á variábel externa.
- *modifiable*, indica se o valor da variábel externa pode ser modificado ou unicamente accedido dende o modelo.

Asociacións:

- *var*, variábel externa cuxo valor é utilizado no modelo.

---

[39] Representouse o acceso externo mediante o patrón de deseño *Proxy* [67]. A representación gráfica do patrón non coincide coa estándar proposta por UML debido a limitacións da ferramenta de modelado utilizada.

Estado de acción ("ActionState")

Representa o estado de activación dunha acción utilizada no modelo. O atributo herdado *value* é de tipo booleano e a visibilidade desta metaclase non pode ser *public* para evitar que o estado das accións sexa modificado externamente.

Estado de etapa ("StepState")

Representa as variábeis internas ao modelo que almacenan o estado das etapas. O atributo herdado *value* é de tipo booleano e o valor inicial será *true* para as etapas iniciais e *false* para as demais. A visibilidade desta metaclase non pode ser *public* para evitar que o estado dun grafcet sexa modificado externamente.

Valor de temporizador ("TimerValue")

A metaclase *TimerValue* representa os valores de temporizadores utilizados nun modelo. A metaclase derívase de *ProcessValue* para indicar que o seu valor é utilizado internamente, mais que o manexo dos temporizadores e o medio de acceso ao seu valor é unha cuestión de implementación non recollida no metamodelo.

Situación ("Situation")

Unha situación é un agrupamento dos estados dun conxunto de etapas.

Condición booleana do modelo ("GrafcetBooleanCondition")

Esta metaclase representa unha expresión booleana na que poden utilizarse eventos e os valores das variábeis do modelo.

Asociacións:

- *vars*, variábeis cuxos valores son utilizados na condición.
- *events*, eventos utilizados na condición.

Condición temporal do modelo ("GrafcetTemporizedCondition")

Representación dunha expresión booleana na que poden utilizarse eventos e os valores das variábeis do modelo e que unicamente é avaliada no intervalo temporal delimitado polos valores dos atributos *delay* e *limit*.

Atributos:

- *delay*, expresión temporal que indica a demora antes da que a expresión non pode ser avaliada. O valor resultado da avaliación da condición non é tido en conta antes de transcorrer o período de tempo indicado polo valor deste atributo.
- *limit*, expresión temporal que indica o límite temporal despois do cal a expresión non pode ser avaliada. O valor resultado da avaliación da condición non é tido en conta unha vez transcorrido o período de tempo indicado polo valor deste atributo.

5.1.2.2. O paquete *Grafcet Core*

Neste paquete defínense as metaclases que representan os elementos sintácticos básicos do Grafcet (Figura 5.5) e os agrupamentos destes elementos en estructuras mais complexas como macroetapas e grafcets conexos (Figura 5.6). A descrición detallada das metaclases deste paquete é a seguinte:

Nodo do modelo ("GrafcetNode")

Esta metaclase representa os nodos que forman parte dun modelo Grafcet, xa sexan etapas ou transicións. Un nodo pode participar nunha ou mais secuencias de control, é dicir, pode conectarse a múltiples nodos que o precedan ou sucedan nas secuencias nas que participe. Esta é unha metaclase abstracta.

Atributos:

- *links*, indica a posición que pode ocupar o nodo nunha secuencia de control (se pode ter predecesores e/ou sucesores).

Asociacións:

- *predecessor*, indica os predecesores do nodo nas secuencias de control nas que este participa.
- *successor*, indica os sucesores do nodo nas secuencias de control nas que este participa.

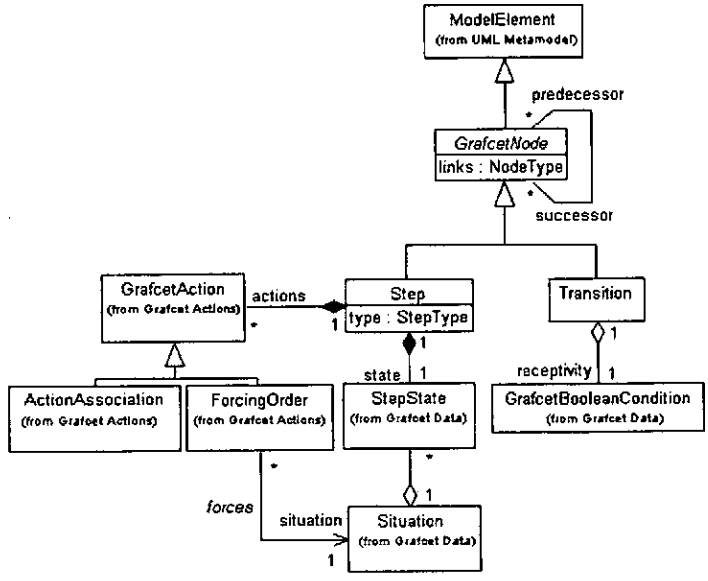


Figura 5.5. Diagrama de clases do paquete *Grafcet Core*: elementos básicos.

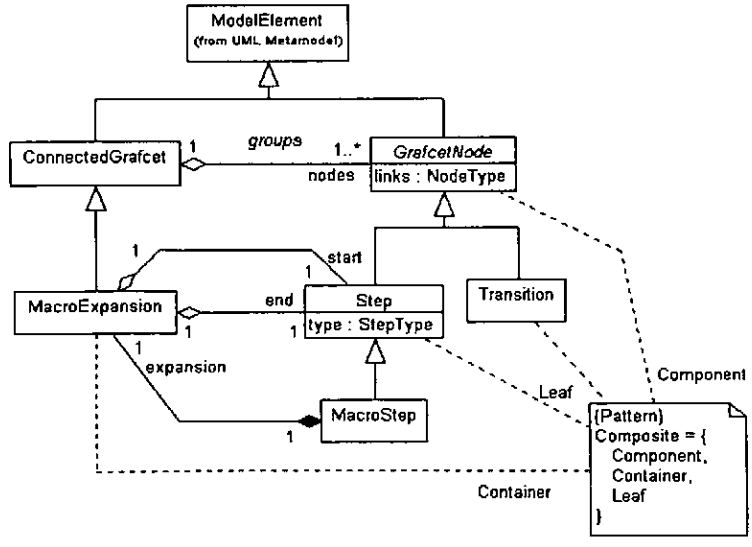


Figura 5.6. Diagrama de clases do paquete *Grafcet Core*: elementos xerárquicos.

Etapa ("Step")

Unha etapa representa un punto da secuencia de control susceptíbel de realizar algunha actuación que modifique o estado do modelo. En cada instante da execución unha etapa pode estar activa ou non. Unicamente as etapas activas poden realizar actuacións. O conxunto de etapas activas determina a situación actual do modelo.

Unha etapa pode realizar múltiples actuacións que modifiquen o estado do modelo (e indirectamente o do sistema controlado polo modelo).

O estado dunha etapa é almacenado nunha variábel booleana do modelo que pode ser utilizada nas condicións lóxicas. Unha etapa pode participar nunha ou mais secuencias de control. Poden conectarse a unha etapa múltiples transicións que a precedan ou a sucedan nas secuencias nas que participe, mais non pode conectarse a outra etapa.

As etapas iniciais e as macroetapas son tipos especiais de etapas. O estado das primeiras é activado ao comezo da interpretación do modelo. O conxunto de etapas iniciais determina a situación inicial dun modelo. En canto ás macroetapas, son representacións abreviadas de partes dun grafcet que cumpren unhas regras sintácticas específicas.

Atributos:

- *type*, indica o tipo de etapa.

Asociacións:

- *state*, indica a variábel do modelo que contén o valor do estado de activación da etapa.
- *actions*, actuacións realizadas pola etapa cando está activa.

Transición ("Transition")

Unha transición representa un punto da secuencia de control no que a execución se detén ate que se cumpra unha condición determinada.

Unha transición pode participar nunha ou mais secuencias de control. Poden conectarse a unha transición múltiples etapas que a precedan ou a sucedan nas secuencias nas que participe, mais non pode conectarse a outra transición. Tampouco é correcto que unha transición non participe nunha secuencia de control (teña un valor *none* no atributo herdado *links*).

Asociacións:

- *receptivity*, condición lóxica que debe cumprirse para continuar a execución da secuencia de control. As regras de evolución do Grafcet (§3.3.1) indican que o resultado da avaliación desta condición só se ten en conta cando todas as etapas que preceden á transición estean activas, excepción feita das transicións fonte (as que teñen o valor *source* no atributo herdado *links*) que están sempre validadas.

Macroetapa ("MacroStep")

Unha macroetapa é un tipo específico de etapa que representa de forma abreviada un conxunto conectado de etapas e transicións denominado macroexpansión da macroetapa. Unha macroetapa pode aparecer nun grafcet na mesma posición que calquera outra etapa e o seu estado dependerá do estado das etapas da súa macroexpansión. As macroetapas non teñen actuacións asociadas.

No metamodelo representouse como unha metaclase derivada de *Step* para modelar así as asociacións e restriccións específicas que non comparte cos demais tipos de etapas. O valor do atributo herdado *type* ten que ser igual a *macro* nesta metaclase. Nótese que para modelar o

feito de que unha macroetapa é un nodo que contén outros nodos na súa macroexpansión, aplicouse o patrón de deseño *Composite* [67].

Asociacións:

- *expansion*, indica o grafcet conexo que representa a macroexpansión da macroetapa.

Grafcet conexo ("ConnectedGrafcet")

Un grafcet conexo é unha agrupación de nodos na que todos están conectados entre si, directa ou indirectamente a través doutros nodos. Cada nodo dun modelo unicamente pode pertencer a un grafcet conexo.

Asociacións:

- *nodes*, nodos agrupados no grafcet conexo.

Macroexpansión ("MacroExpansion")

Unha macroexpansión é un tipo de grafcet conexo que modela os contidos dunha macroetapa. As macroexpansións teñen unha restricción sintáctica, teñen que comezar e rematar por unha única etapa, de xeito que a substitución dunha macroetapa pola súa macroexpansión manteña a corrección sintáctica do modelo.

Asociacións:

- *start*, indica a etapa de comezo da macroexpansión.
- *end*, indica a etapa final da macroexpansión.

5.1.2.3. O paquete *Grafcet Actions*

No paquete *Grafcet Actions* defínense as metaclases que permiten modelar as actuacións asociadas ás etapas dun Grafcet: accións e ordes de forzado. A Figura 5.7 mostra o diagrama de clases deste paquete e a relación entre as metaclases definidas e as metaclases UML comentadas en (§5.1.1.1). A descrición detallada das metaclases deste paquete é a seguinte:

Acción do modelo ("GrafcetAction")

Esta metaclase é a base de todas as actuacións que poden realizarse nunha etapa dun grafcet. Actualmente hai definidos dous tipos de actuacións: as accións e as ordes de forzado. Calquera outra actuación que poida definirse no futuro incluiríase no metamodelo como derivada desta metaclase. Esta é unha metaclase abstracta.

Asociación ("ActionAssociation")

Derivada da metaclase *GrafcetAction*, representa os bloques de acción asociados a unha etapa según o definido no estándar [86]. Cada bloque de acción IEC (§3.6.1.3) ten catro partes: un nome, un indicador, un cualificador e unha implementación. O nome e o indicador son variábeis booleanas modificadas polo bloque durante a súa execución, e o cualificador indica a semántica da súa execución. No metamodelo inclúense ademais algunhas características adicionais non definidas polo IEC, como os bloques condicionais ou as accións internas.

Atributos:

- *type*, cualificador do bloque de acción. Os cualificadores definidos son: N, R, S, P, P1 e P0. Os cualificadores IEC: L, D, SD, DS e SL son representados no metamodelo mediante unha condición temporal asociada á metaclase.

- *internal*, indica se un bloque de acción debe ou non ser executado en situacións inestábeis durante as evolucións internas do modelo. O valor deste atributo unicamente ten validez en grafcets interpretados mediante un algoritmo de evolución tipo ARS (§3.3.2.2). O valor deste atributo non pode ser *true* para bloques con cualificador tipo N, xa que internamente só poden executarse accións impulsionais.

Asociacións:

- *name*, indica a variábel booleana do modelo modificada polo bloque de acción durante a súa execución, dacordo á semántica indicada polo cualificador do bloque. A variábel non pode ser un valor de entrada do proceso.
- *indicator*, indica a variábel booleana do modelo modificada polo bloque de acción para indicar a finalización da súa execución. A variábel non pode ser un valor de entrada do proceso.
- *condition*, indica unha condición temporal que modifica a semántica de execución do bloque de acción. Permite demorar, condicionar e limitar os efectos da execución dunha acción.

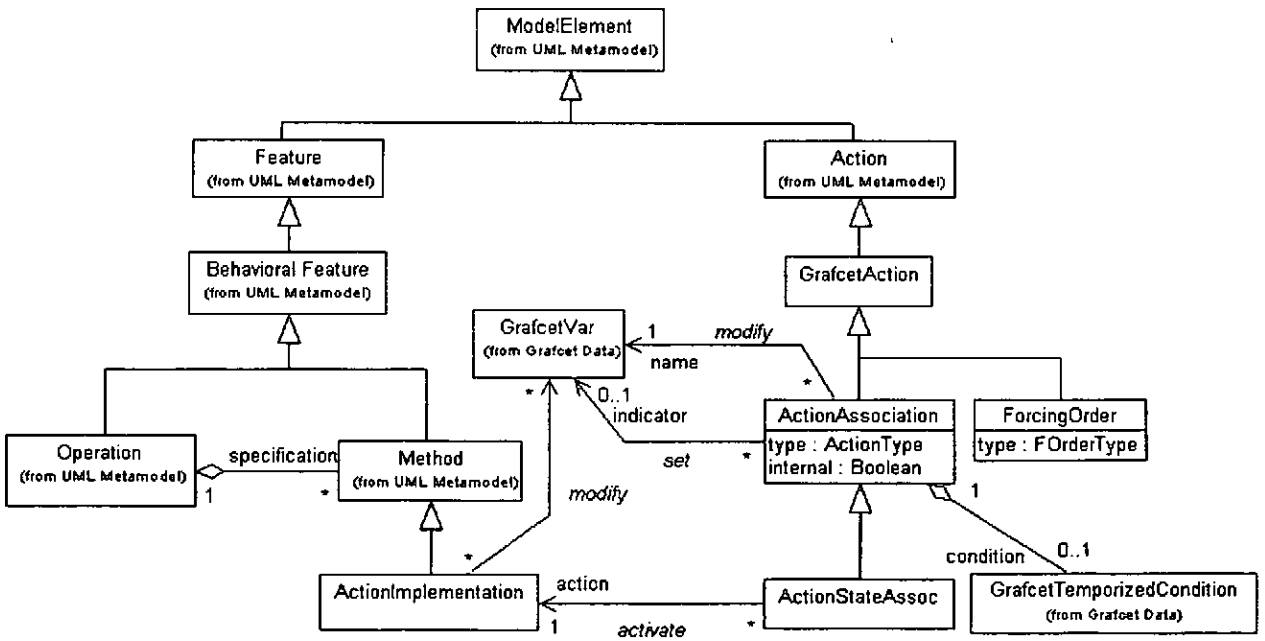


Figura 5.7. Diagrama de clases do paquete *Grafcet Actions*.

Bloque de acción ("ActionStateAssoc")

Metaclase que representa un tipo de bloque de acción que modifica unha variábel de estado de acción —*ActionState* (§5.1.2.1)—. O estado da variábel modificada indica o estado de activación da acción e mediante a súa modificación controlase a execución do código da acción.

Asociacións:

- *action*, indica a acción cuxa execución é controlada polo bloque de acción mediante a modificación do valor da variábel do modelo que almacena o seu estado de activación.

Código de acción ("ActionImplementation")

Esta metaclase representa a implementación dunha acción. Deriva da metaclase UML *Method*, polo que pode representar a implementación de operacións definidas ou herdadas por un grafcet. O medio utilizado para a implementación da acción non é especificado polo metamodelo.

Asociacións:

- *modify*, indica as variábeis do modelo accedidas (e posiblemente modificadas) dende a implementación da acción.

Orde de forzado ("ForcingOrder")

Metaclase que representa as ordes de forzado dacordo ao definido en [2]. No metamodelo cada orde de forzado relaciona un par de grafcets parciais (Figura 5.8), o 'forzador' e o forzado, e especifica a situación do grafcet forzado mentres a orde estea activa.

Atributos:

- *type*, indica o tipo da orde de forzado.

Asociacións:

- *forcerOrder*, indica o grafcet parcial que contén a etapa á que a orde de forzado está asociada, é dicir, o grafcet 'forzador'.
- *forcingOrder*, indica o grafcet parcial cuxa situación é forzada.
- *situation*, indica a situación forzada.

5.1.2.4. O paquete *Grafcet Hierarchy*

No paquete *Grafcet Hierarchy* (Figura 5.8) defínense as metaclases e relacións que permiten modelar as estructuras xerárquicas do Grafcet: a estructural (§3.2.2.3) e a de forzado (§3.2.2.4), así como o contexto dun modelo grafcet dentro dun modelo UML. A descrición detallada das metaclases deste paquete é a seguinte:

Grafcet global ("GlobalGrafcet")

Esta metaclase representa unha abstracción do comportamento do sistema modelado e constitúe o elemento de máis alto nivel na xerarquía estructural do modelo Grafcet. O contexto é o mesmo que o dos StateCharts no metamodelo UML, polo que na práctica o metamodelo proposto permite utilizar o Grafcet como alternativa a aqueles para a especificación do comportamento dos elementos dinámicos dun modelo UML. Neste nivel están definidas as variábeis e accións utilizadas no modelo, e agrúpanse os grafcets parciais que definen a súa estructura.

Asociacións:

- *context*, indica o elemento do modelo cuxo comportamento é especificado polo grafcet global.
- *actions*, agrupamento das accións implementadas no grafcet global.
- *data*, agrupamento das variábeis definidas no grafcet global.
- *subsystems*, agrupamento dos grafcets parciais que definen a estructura do grafcet global.

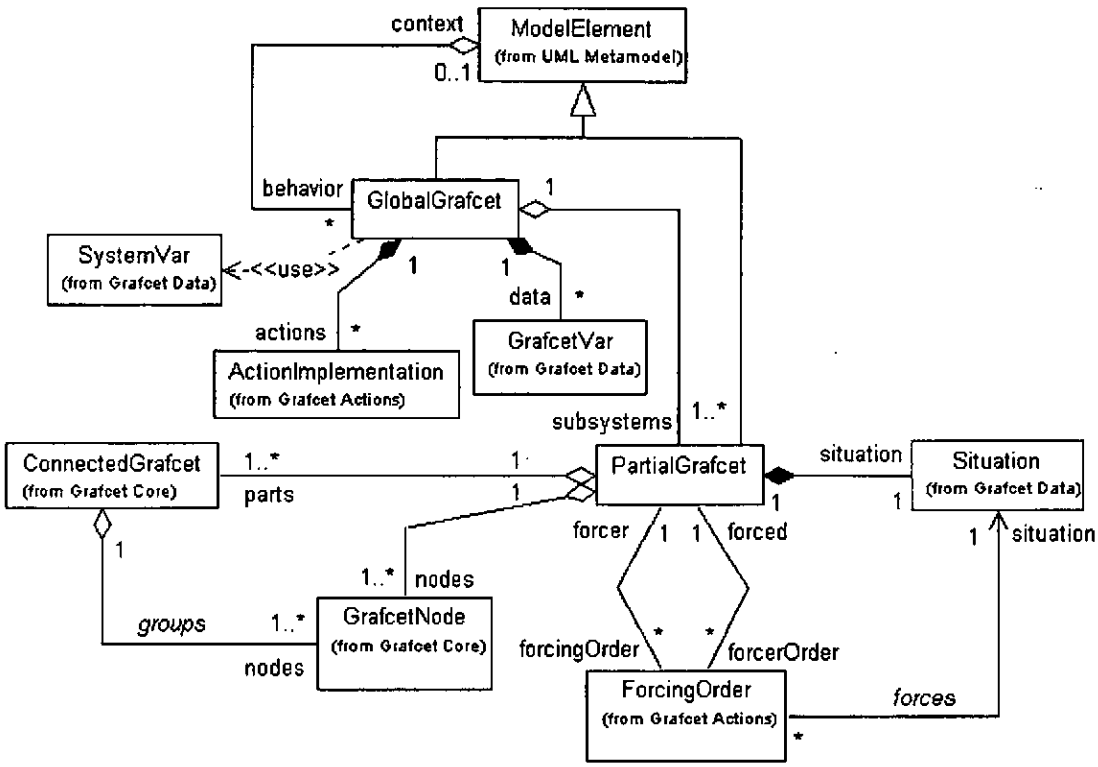


Figura 5.8. Diagrama de clases do paquete *Grafcet Hierarchy*.

Grafcet parcial ("PartialGrafcet")

Esta metaclase representa unha abstracción do comportamento dunha parte identificada do elemento modelado. Na práctica constitúe un medio de reducir a complexidade do modelo dividíndoo en partes que poden ou non interactuar entre elas e que se organizan xerarquicamente mediante as relacións de forzado.

As metaclases *PartialGrafcet* e *ConnectedGrafcet* son as que permiten representar a xerarquía estructural dos modelos Grafcet. O nivel mais baixo da xerarquía está representado polos nodos do modelo que se agrupan en grafcets conexos, que a súa vez forman grafcets parciais. O conxunto de grafcets parciais define o comportamento dun grafcet global (e polo tanto do elemento modelado).

Debido a que na práctica a utilización dos grafcets conexos como partes identificadas dun modelo non é habitual, coa excepción das macroexpansións, no metamodelo proposto representouse a relación entre o grafcet parcial e os nodos que o forman por duplicado: mediante a asociación directa entre as metaclases *PartialGracet* e *GrafcetNode,* e mediante a asociación indirecta entre estas dúas clases a través da metaclase *ConnectedGrafcet*. Deste xeito o usuario do metamodelo pode elixir se utilizar ou non os grafcets conexos como elementos significativos nos seus modelos.

Asociacións:

- *nodes*, agrupamento dos nodos que pertencen ao grafcet parcial. Cada nodo está contido tamén nun dos grafcets conexos que forman parte do grafcet parcial.
- *situation*, indica a situación actual do grafcet parcial, que é o conxunto de valores dos estados de activación das etapas que forman parte do grafcet parcial. A situación dun grafcet global está determinada polas situacións dos grafcets parciais que o forman.

- *parts*, agrupamento dos grafcets conexos que son parte dun grafcet parcial.
- *forcer* e *forced*, especifican a relación xerárquica establecida entre dous grafcets parciais a través dunha orde de forzado. Esta xerarquía ten que ser acíclica e non está permitido o forzado múltiple simultáneo dun grafcet parcial durante a execución do modelo.

5.1.3. Semántica estática do modelo

Nesta sección complétase a sintaxe abstracta do metamodelo cun conxunto de invariantes expresadas utilizando a linguaxe OCL. Estas invariantes expresan a semántica estática que as instancias do metamodelo deben cumprir para formar un modelo sintacticamente correcto.

5.1.3.1. Cálculo do peche transitivo

No metamodelo faise preciso expresar algunhas regras que comproben que unha instancia dunha metaclase non está relacionada recursivamente consigo mesma (p.e. para indicar que a xerarquía de forzado non pode conter ciclos). Precísase polo tanto dun medio para calcular o peche transitivo dunha relación (asociación) a partir dunha instancia dunha metaclase dada. OCL non especifica ningún operador ou operación para realizar este cálculo [113], polo que foi preciso incluílo como parte do metamodelo proposto. A descrición xenérica do cálculo do peche transitivo é a seguinte:

```
T→transitiveClosure(closure : Set(T)) : Set(T);
transitiveClosure = if closure→includesAll(closure→allAdjacents)
                      then closure
                      else transitiveClosure(closure→union(closure→allAdjacents))
                    endif
```

O cálculo faise recursivamente, incluíndo os adxacentes dos elementos xa incluídos no peche ate que non haxa mais adxacentes que incluír. Este cálculo evita a recursividade infinita en caso de ciclos, pois detense unha vez que todos os elementos adxacentes (directa ou indirectamente) foron xa incluídos no peche. O cálculo a partir dunha instancia podería facerse do xeito seguinte:

```
allRelated(t:T) : Set(T);
allRelated = transitiveClosure(Set{t})
```

A relación utilizada para o cálculo do peche transitivo é a adxacencia entre instancias de tipo T. Presuponse no tipo T a existencia dunha operación que devolva os adxacentes dun conxunto de instancias (almacenados nun atributo denominado *adjacents*). A descrición desta operación é a seguinte:

```
T→allAdjacents(s : Set(T)) : Set(T);
allAdjacents = s→collect(adjacents)→asSet
```

Nas regras do metamodelo que utilizan a operación *transitiveClosure* indicouse mediante un comentario a relación utilizada para o cálculo do peche transitivo. A relación así indicada terá a mesma funcionalidade que a que se acaba de describir para a adxacencia. Deste xeito evítase a necesidade de redefinir a operación de cálculo do peche transitivo para cada metaclase ou relación específica.

5.1.3.2. Regras semánticas

Código de Acción ("ActionImplementation")

1. Unha acción non pode modificar o estado dunha etapa, o estado doutra acción ou o valor dunha variábel externa non modificábel.

```
self.GrafcetVar→select(var |
    ((var.oclIsTypeOf(StepState) or
    var.oclIsTypeOf(ActionState) ) and var.changeability = #frozen) or
    (var.oclIsTypeOf(ProcessValue) and
      var.oclAsType(ProcessValue).modifiable = false))→isEmpty
```

Asociación ("ActionAssociation")

1. As variábeis modificadas pola asociación son booleanas.

```
-- nome
self.name.value.oclIsTypeOf(Boolean)
-- indicador
self.indicator→notEmpty implies self.indicator.value.oclIsTypeOf(Boolean)
```

2. As variábeis modificadas pola asociación están definidas no mesmo grafcet global.

```
-- nome
self.Step.PartialGrafcet.GlobalGrafcet.data→includes(self.name)
--indicador
self.indicator→notEmpty implies
  self.Step.PartialGrafcet.GlobalGrafcet.data→includes(self.indicator)
```

3. As variábeis accedidas pola condición da asociación están definidas no mesmo grafcet global.

```
self.condition→notEmpty implies
  self.Step.PartialGrafcet.GlobalGrafcet.data→includesAll(self.condition.allVars)
```

4. Unicamente as accións impulsionais poden ser internas.

```
self.internal = true implies self.type <> #N
```

5. As accións internas non poden ter condicións temporizadas (demoradas ou limitadas).

```
self.internal = true and self.condition→notEmpty implies
  self.condition→delay.body = '' and self.condition→limit.body = ''
```

Bloque de acción ("ActionStateAssoc")

1. A acción activada polo bloque de acción pertence ao mesmo grafcet global que o bloque.

```
self.Step.PartialGrafcet.GlobalGrafcet.actions→includes(self.action)
```

Condición booleana do modelo ("GrafcetBooleanCondition")

Operacións auxiliares

1. *allVars*, calcula todas as variábeis accedidas dende unha condición booleana, xa sexa directamente ou a través dun evento.

allVars : Set(GrafcetVar);
allVars = self.GrafcetVar→union(self.Event->collect(var)→asSet)

Estado de acción ("ActionState")

1. As variábeis que conteñen os valores do estado de activación das accións son de tipo booleano.

self.value.oclIsTypeOf(Boolean)

Estado de etapa ("StepState")

1. As variábeis que conteñen os valores do estado de activación das etapas son de tipo booleano.

self.value.oclIsTypeOf(Boolean)

Etapa ("Step")

1. Os antecesores dunha etapa só poden ser transicións.

self.predecessor→forAll(oclIsTypeOf (Transition))

2. Os sucesores dunha etapa só poden ser transicións.

self.successor→forAll(oclIsTypeOf (Transition))

Evento ("Event")

1. Os eventos son cambios de estado de variábeis booleanas.

self.var.value.oclIsTypeOf(Boolean)

Grafcet conexo ("ConnectedGrafcet")

1. Todos os nodos agrupados nun grafcet conexo están conectados directa ou indirectamente entre eles.

```
self.nodes→forAll(GrafcetNode node1, node2 |
        node1 <> node2 implies
            (node1.allSuccessors→includes(node2)
              or node1.allPredecessors→includes(node2)))
```

Grafcet global ("GlobalGrafcet")

1. O contexto dun grafcet global pode ser ou un clasificador ou unha característica dinámica[40].

```
self.context.notEmpty implies
  (self.context.oclIsKindOf(BehavioralFeature) or
     self.context.oclIsKindOf(Classifier))
```

2. Os grafcets parciais relacionados mediante unha orde de forzado pertencen ao mesmo grafcet global.

[40] Esta regra especifica que o contexto dos modelos Grafcet é o mesmo que o definido en UML para os StateCharts.

self.subsystems→forAll(PartialGrafcet pg |
self.subsystems→includesAll(pg.allForced) and
self.subsystems→includesAll(pg.allForcers))

3. A xerarquía de forzado é acíclica.

self.subsystems→forAll(PartialGrafcet pg |
not(pg.allForcers→includes(pg))

4. Os identificadores dos grafcets globais son únicos.

GlobalGrafcet.allInstances→forAll(g |
g <> self implies g.name <> self.name)

5. Os identificadores dos grafcets parciais incluídos nun grafcet global son únicos.

self.subsystems→forAll(PartialGrafcet pg1, pg2 |
pg1 <> pg2 implies pg1.name <> pg2.name)

6. Os identificadores das variábeis incluídas nun grafcet global son únicos.

self.data→forAll(GrafcetVar var1, var2 |
var1 <> var2 implies var1.name <> var2.name)

7. Os identificadores das accións incluídas nun grafcet global son únicos.

self.actions→forAll(ActionImplementation act1, act2 |
act1 <> act2 implies act1.name <> act2.name)

8. Para cada acción incluída nun grafcet global hai unha e só unha variábel co mesmo nome que almacena o seu valor de activación.

self.actions→forAll(action |
self.data→select(name = action.name and isOclType(ActionState))→size = 1))

9. Para toda variábel incluída nun grafcet global que almacene o valor do estado de activación dunha acción, hai unha e só unha acción.

self.data→forAll(GrafcetVar var|
var.oclIsTypeOf(ActionState) implies self.actions→select(name = var.name)→size = 1)

10. Para toda variábel incluída nun grafcet global que almacene o estado de activación dunha etapa, hai unha e só unha etapa.

self.data→forAll(GrafcetVar var|
var.oclIsTypeOf(StepState) implies self.subsystems.nodes→select(name = var.name)→size = 1)

11. As variábeis modificadas polas accións incluídas nun grafcet global están definidas tamén no mesmo grafcet global.

self.actions→forAll(action |
self.data→includesAll(action→GrafcetVar))

Grafcet parcial ("PartialGrafcet")

1. Os identificadores dos grafcets conexos incluídos nun grafcet parcial son únicos.

```
self.parts→forAll(ConnectedGrafcet cg1, cg2 |
    cg1 <> cg2 implies cg1.name <> cg2.name)
```

2. Os identificadores de nodos de igual tipo nun grafcet parcial son únicos.

```
self.nodes→forAll(GrafcetNode node1, node2 |
    (node1 <> node2 and node1.type = node2.type)
        implies node1.name <> node2.name)
```

3. Cada nodo dun grafcet parcial está incluído nun (e só nun) dos grafcets conexos que inclúe.

```
self.nodes→forAll(node |
    self.parts→select(nodes→includes(node))→size = 1)
```

4. Todos os nodos dun grafcet conexo están incluídos tamén no grafcet parcial que o contén.

```
self.parts→forAll(cg |
    self.nodes→includesAll(cg→nodes))
```

5. Para cada etapa dun grafcet parcial hai unha variábel no grafcet global que almacena o seu estado de activación.

```
self.node→select(isOclType(Step))→forAll(step |
    self.GlobalGrafcet.data→includes(step.state))
```

6. A situación dun grafcet parcial inclúe unicamente variábeis de estado de etapas pertencentes ao grafcet parcial.

```
self.node.includesAll(self.situation.allSteps)
```

Operacións auxiliares

1. *allForced*, calcula o peche transitivo dos grafcets forzados directa ou indirectamente por un grafcet parcial dado (o conxunto de grafcets parciais que pertencen a un nivel inferior da xerarquía de forzado).

```
allForced : Set(PartialGrafcet);  -- utiliza a relación forcingOrder.forced (Figura 5.8)
allForced = transitiveClosure(forcingOrder.forced→asSet)
```

2. *allForcers*, calcula o peche transitivo dos grafcets que forzan directa ou indirectamente a un grafcet parcial dado (o conxunto de grafcets parciais que pertencen a un nivel superior da xerarquía de forzado).

```
allForcers: Set(PartialGrafcet);  -- utiliza a relación forcerOrder.forcer (Figura 5.8)
allForcers = transitiveClosure(forcerOrder.forcer→asSet)
```

Macroetapa ("MacroStep")

1. O valor do atributo herdado *type* é sempre *macro* nunha macroetapa.

```
self.type = #macro
```

2. As macroetapas non teñen accións asociadas.

```
self.actions->isEmpty
```

3. Os tipos de etapas inicial e final dunha macroexpansión están restrinxidos polo tipo da macroetapa á que pertencen.

```
self.links = #none     -- macroetapa fonte e sumidoiro
    implies (self.expansion.start.links = #source
      and self.expansion.end.links = #target)
self.links = #target   -- macroetapa sumidoiro
    implies (self.expansion.start.links = #both
      and self.expansion.end.links = #target)
self.links = #source   -- macroetapa fonte
  implies (self.expansion.start.links = #source
    and self.expansion.end.links = #both)
self.links = #both     -- macroetapa
  implies (self.expansion.start.links = #both
    and self.expansion.end.links = #both)
```

Macroexpansión ("MacroExpansion")

1. As etapas inicial e final dunha macroexpansión son diferentes.

```
self.start <> self.end
```

2. A etapa inicial non pode ser unha etapa sumidoiro.

```
self.start.links = #both or
self.start.links = #source
```

3. A etapa final non pode ser unha etapa fonte.

```
self.end.links = #both or
self.end.links = #target
```

Nodo do modelo ("GrafcetNode")

1. Multiplicidade dos predecesores e sucesores dun nodo en función do valor do atributo *links*.

```
node.links = #none     -- nodo fonte e sumidoiro
  implies (node.predecessor→size = 0
    and node.succesor→size = 0)
node.links = #target   -- nodo sumidoiro
  implies (node.predeccesor→size > 0
    and node. succesor→size = 0)
node.links = #source   -- nodo fonte
  implies (node. predeccesor→size = 0
    and node. succesor→size > 0)
node.links = #both     -- nodo normal
  implies (node. predeccesor→size > 0
    and node. succesor→size > 0)
```

2. Todos os nodos conectados entre si están incluídos no mesmo grafcet conexo.

```
self.ConnectedGrafcet.nodes→includesAll(self.allSuccessors) and
self.ConnectedGrafcet.nodes→includesAll(self.allPredecessors)
```

Operacións auxiliares

1. *allPredecessors*, calcula o peche transitivo dos predecesores dun nodo: o conxunto de todos os nodos que o preceden na secuencia de control.

   allPredecesors : Set(GrafcetNode); -- utilízase a relación predecessor (Figura 5.5)
   allPredecesors = transitiveClosure(self→predecessor→asSet)

2. *allSuccessors*, calcula o peche transitivo dos sucesores dun nodo: o conxunto de todos os nodos que o suceden na secuencia de control.

   allSuccesors : Set(GrafcetNode); -- utilízase a relación successor (Figura 5.5)
   allSuccesors = transitiveClosure(self→successor→asSet)

Orde de forzado ("ForcingOrder")

1. As macroetapas non poden ser forzadas (o forzado debe facerse utilizando as etapas incluídas na macroexpansión da macroetapa).

   self.situation.allSteps→forAll(step.type <> #macro)

2. Todas as etapas forzadas están contidas no grafcet parcial forzado.

   self.forced.nodes→includesAll(self.situation.allSteps)

3. A etapa á que a orde de forzado está asociada está contida no grafcet parcial forzador.

   self.forcer.nodes→includes(self.step)

Situación ("Situation")

Operacións auxiliares

1. *allSteps*, devolve o conxunto de etapas cuxos estados de activación son almacenados na situación.

   allSteps : Set(Step);
   allSteps = self.StepState→collect(step)→asSet

Transición ("Transition")

1. As transicións non poden ser fonte e sumidoiro ao mesmo tempo.

   self.links <> #none

2. Os antecesores dunha transición só poden ser etapas.

   self.predecessor→forAll(oclIsKindOf(Step))

3. Os sucesores dunha transición só poden ser etapas.

   self.successor→forAll(oclIsKindOf(Step))

4. As variábeis accedidas pola receptividade están definidas no mesmo grafcet global.

   self.PartialGrafcet.GlobalGrafcet.data→includesAll(self.receptivity.allVars)

5.2. Implementación do metamodelo

Para a utilización práctica do metamodelo proposto desenvolveuse unha libraría C++ que implementa as metaclases e asociacións nel definidas ao tempo que garante certos aspectos da corrección sintáctica durante a construcción dos modelos (p.e. a identificación, a alternancia etapa-transición, etc.). Ademais a libraría proporciona medios para realizar operacións como a comprobación da coherencia da xerarquía de forzado ou a substitución de macroetapas.

5.2.1. Funcionalidade básica

A libraría está implementada partindo dun conxunto de clases que ofrecen unhas funcionalidades básicas: contedores e iteradores, representación de variábeis e tipos de datos, identificación, 'persistencia', etc. Algunhas destas funcionalidades son comúns a outras partes da ferramenta proposta nesta tese de doutoramento e outras foron deseñadas para resolver cuestións específicas relacionadas coa implementación do metamodelo. Neste apartado descríbense estas últimas. Poden consultarse no Anexo B algunha das funcionalidades comúns con outras partes da ferramenta.

5.2.1.1. Identificación

Relacionadas coa identificación dos elementos dun Grafcet foron dous os problemas principais que houbo que resolver:

1. A xeración automática de identificadores únicos.
2. O almacenamento de nodos de diferentes tipos con igual identificador nunha mesma colección.

Para a resolución do primeiro problema definiuse unha clase que funciona como 'repositorio' de identificadores numéricos clasificados en categorías. Esta clase leva para cada categoría o rexistro dos identificadores que están sendo utilizados e proporciona o identificador numérico máis baixo dos que estean libres. Para reducir a memoria necesaria, o rexistro de identificadores é almacenado en forma de intervalos. No diagrama de clases da Figura 5.9 móstranse as clases e relacións definidas.

O código seguinte mostra un exemplo da utilización do repositorio na obtención de identificadores únicos para os nodos dun grafcet:

```
1. IdRepository ids;
2. ids.addCategory('Step');
3. ids.addCategory('Trans');
4. string step0 = ids.lockId('Step');          // reserva o primeiro libre (1)
5. string step10 = ids.lockId('Step', '10');   // reserva a etapa 10
6. string step20 = ids.lockId('Step', '20');   // reserva a etapa 20
7. bool isused = ids.isIdLocked('Step', '1');  // devolve true
8. string trans0 = ids.lockId('Trans');        // reserva o primeiro libre (1)
9. string trans1 = ids.lockId('Trans');        // reserva o primeiro libre (2)
10. string trans10 = ids.lockId('Trans', '10'); // reserva a transición 10
11. string trans10 = ids.lockId('Trans', '10'); // o 10 xa está reservado
12. ids.unlockId('Trans', '10');               // libera a transición 10
13. trans10 = ids.lockId('Trans', '10');       // reserva a transición 10
```

O segundo problema aparece cando queren almacenarse nunha mesma colección múltiples nodos Grafcet de diferentes tipos (etapas, transicións, ...). As coleccións utilizadas (§B.1) presupoñen que os elementos almacenados teñen un identificador alfanumérico único, mais isto non se cumpre para os nodos dun grafcet, pois é posíbel que unha etapa e unha transición teñan

o mesmo identificador. En consecuencia foi preciso definir un mecanismo que permitira utilizar dous identificadores para cada obxecto:

1. Un identificador alfanumérico único para os obxectos dun tipo dado (este é o identificador proporcionado polo usuario durante a definición dos modelos).
2. Un identificador cualificado único independente do seu tipo para todos os obxectos (este é o identificador utilizado internamente nas coleccións de obxectos).

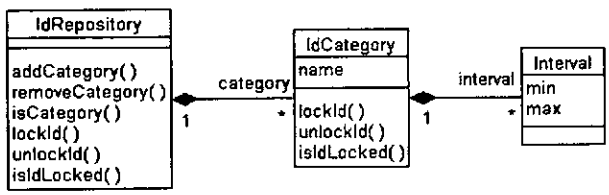


Figura 5.9. Diagrama de clases do repositorio de identificadores.

A relación entre ambos identificadores debe ser moi estreita, de xeito que o cálculo dun a partir do outro sexa directo e que a modificación dun actualice o outro automaticamente. O diagrama de clases da Figura 5.10 mostra como se resolveu este problema.

A clase abstracta *IdContainer* define a interface que teñen que implementar as clases que incluían o mecanismo de dobre identificación. A clase *Identifier* é a que representa o identificador proporcionado polo usuario[41]. Esta clase define as operacións básicas para realizar comparacións, modificar e almacenar o identificador. A clase *IdentifiedClass* é utilizada como exemplo de como incluír a dobre identificación nunha clase calquera. Nesta clase o atributo *key* almacena o identificador utilizado internamente nas coleccións de obxectos e os operadores de comparación definidos están implementados utilizando o valor deste atributo. Na implementación da libraría cúmprese a relación *key* = *Class_Name.identifier*, polo que a partir do seu valor é inmediato obter o identificador e viceversa.

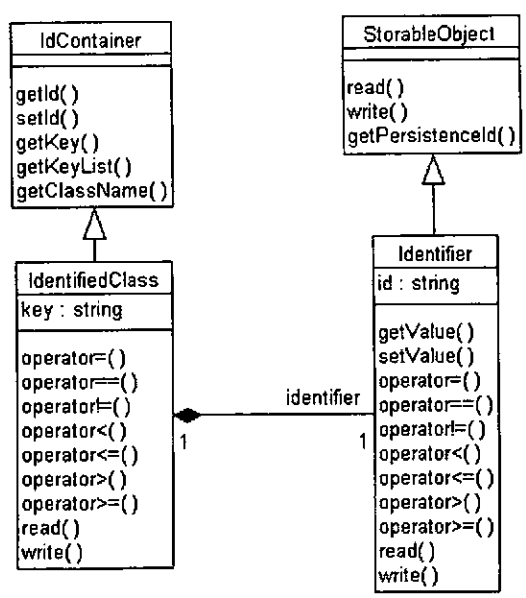


Figura 5.10. Diagrama das clases implicadas no mecanismo de identificación.

[41] Esta clase é derivada da clase *StorableObject*, que é a que proporciona o soporte á 'persistencia' na libraría implementada.

O código seguinte mostra un exemplo da utilización de identificadores:

```
14. Node node;                          // clase que implementa a dobre identificación
15. node.setId('10');
16. string id = node.getId();           // id = '10'
17. string key = node.getKey();         // key = 'Node.10'
18. string type = node.getClassName(); // type = 'Node'
```

5.2.1.2. Notificacións

Os modelos Grafcet son almacenados en memoria como grafos dirixidos cíclicos con estructura xerárquica. A información da estructura está repartida entre diferentes elementos e a modificación dalgún deles require a actualización da información que conteñen os demais. Por exemplo, a eliminación dun nodo implica que o seu identificador sexa liberado (esta información está no grafcet parcial que contén o nodo), as súas conexións eliminadas (actualización dos nodos adxacentes) e os grafcets conexos actualizados (ao eliminar un nodo o grafcet conexo no que está incluído pode dividirse en dous ou mais grafcets conexos). Para proporcionar esta funcionalidade definiuse un mecanismo de notificacións[42], cando un elemento é modificado e esa modificación require actualizar a información almacenada noutro lugar da estructura, o elemento afectado envía unha notificación aos seus adxacentes indicando o cambio realizado. O elemento da estructura que reciba a notificación pode reenviala, procesala ou simplemente ignorala. Un inconveniente deste mecanismo é a posibilidade dunha explosión combinatoria de notificacións que afecte ao rendemento. Na implementación da libraría Grafcet isto non supuxo un problema, pois as notificacións realízanse ben entre elementos de diferentes niveis xerárquicos, ben entre adxacentes directos do mesmo nivel, polo que a propagación dunha notificación afecta a un número moi reducido de elementos. A Figura 5.11 mostra o diagrama das clases definidas para implementar esta funcionalidade.

As clases que manexen notificacións teñen que implementar a interface NotificationManager, que inclúe operacións para activar e desactivar a recepción de notificacións e un manexador xenérico —método onNotification— que pode ser refinado nas subclases. As notificacións son representadas mediante a clase xenérica Notification da que poden derivarse diferentes tipos específicos de notificacións, ou ben utilizar o atributo info para ese fin.

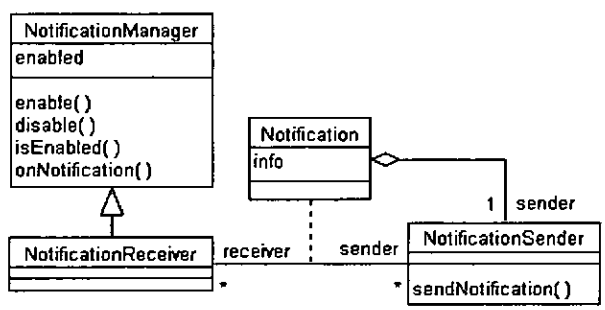


Figura 5.11. Diagrama das clases implicadas no mecanismo de notificación.

[42] Este mecanismo é unha implementación do patrón de deseño *Observer* [67].

5.2.2. Clases da libraría

A Figura 5.12 mostra as clases definidas para implementar a declaración das variábeis utilizadas nos modelos[43]. Todas as declaracións de variábeis utilizadas son instancias de clases obtidas mediante a especialización da clase parametrizada *GescaVarDecl* para diferentes tipos de datos básicos: *SFCBool*, *SFCInt*, etc. As declaracións de variábeis que almacenan os valores de estado de accións e etapas son instancias de clases derivadas de *SFCBool*. Tamén se define un contedor específico —clase *SFCVarDeclarations*— para almacenar conxuntos de declaracións[44].

As clases que implementan as accións e receptividades móstranse no diagrama da Figura 5.13. Por conveniencia definíronse algunhas clases abstractas —*Enunciation*, *Statement*, *Expression* e *InternalOrder*— que permiten organizar conceptualmente as clases definidas. Definiuse tamén un contedor específico para almacenar as accións —clase *SFCActionSeq*—. Nótese que na implementación da libraría agrupáronse as metaclases *ActionAssociation* e *GrafcetTemporizedCondition* nunha única clase —*SFCActionAssociation*—, e que a situación forzada por unha orde de forzado é agregada na clase *SFCForcingOrder* —atributo *situation*—.

O diagrama da Figura 5.14 mostra as clases abstractas que proporcionan os conceptos básicos utilizados para implementar as metaclases dos paquetes *Grafcet Core* (§5.1.2.2) e *Grafcet Hierarchy* (§5.1.2.4) do metamodelo. Como pode verse definense dous contedores específicos para almacenar nodos e grafcets —clases *SFCNodeSeq* e *SFCSeq*, respectivamente—, obtidos por especialización da clase parametrizada *pdcid2ptrSeq*. Ademais destes contedores, as principais clases definidas son:

- *SFCRoot*, clase base de todas as demais, implementa as interfaces *IdContainer*, *pdcptrSeqElem* e *NotificationSender* proporcionando, respectivamente, as seguintes funcionalidades: dobre identificación (§5.2.1.1); persistencia e semántica de asignación por valor (§B.1); e recepción de notificacións (§5.2.1.2).
- *SFCNode*, clase abstracta derivada de *SFCRoot* que implementa a metaclase *GrafcetNode* (§5.1.2.2). As asociacións cos nodos antecesores e sucesores son implementadas mediante os atributos *before* e *after* —de tipo *SFCNodeSeq*—.
- *SFCBase*, clase abstracta derivada de *SFCRoot* que proporciona as definicións comúns a todas as clases que participan na estructuración xerárquica dos modelos Grafcet. As instancias desta clase representan os diferentes niveis da xerarquía estructural. Os grafcets contidos no nivel almacénanse no atributo *sfcs* —de tipo *SFCSeq*—.

  A notificación de cambios entre niveis xerárquicos diferentes impleméntase mediante as clases *SFCChild* e *SFCParent*. Estas clases son unha especialización do mecanismo de notificación (§5.2.1.2) mediante a cal todo elemento do modelo mantén unha referencia ao nivel xerárquico inmediatamente superior. Mediante esta referencia poden propagarse as notificacións dende os niveis inferiores da xerarquía aos superiores. A Táboa 5-I e a Táboa 5-II mostran as notificacións identificadas durante a implementación da libraría e unha descrición da función realizada en cada clase que manexa a notificación.

---

[43] A regra xeral utilizada para identificar as clases da libraria consistiu en tomar os nomes utilizados no metamodelo e precedelos co prefixo *SFC*. En caso de que o nome do metamodelo contivese a palabra Grafcet esta é eliminada na libraría.

[44] Esta clase é unha instanciación da clase parametrizada *pdcid2ptrSeq*, que é un dos contedores que forma parte da libraría común (§B.1).

- *SFCNodeContainer*, clase abstracta derivada de *SFCBase*. Representa os niveis xerárquicos do Grafcet que inclúen nodos. Os nodos incluídos no nivel son almacenados no atributo *nodes* —de tipo *SFCNodeSeq*—.
- *SFCConnectedBase*, clase abstracta derivada de *SFCNodeContainer* que representa os niveis xerárquicos que conteñen nodos e nos que non se leva rexistro dos identificadores utilizados.

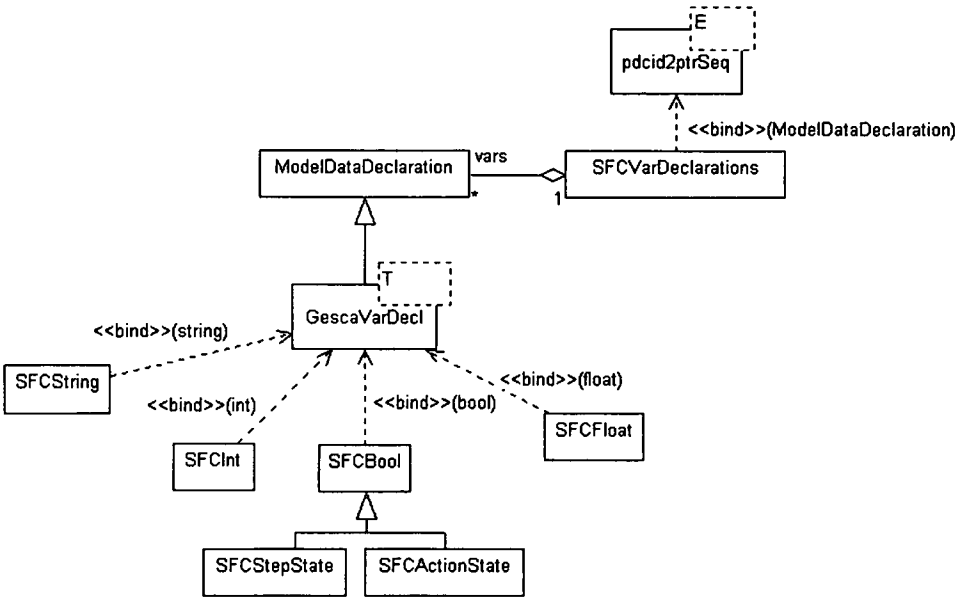


Figura 5.12. Diagrama das clases que implementan a declaración de variábeis.

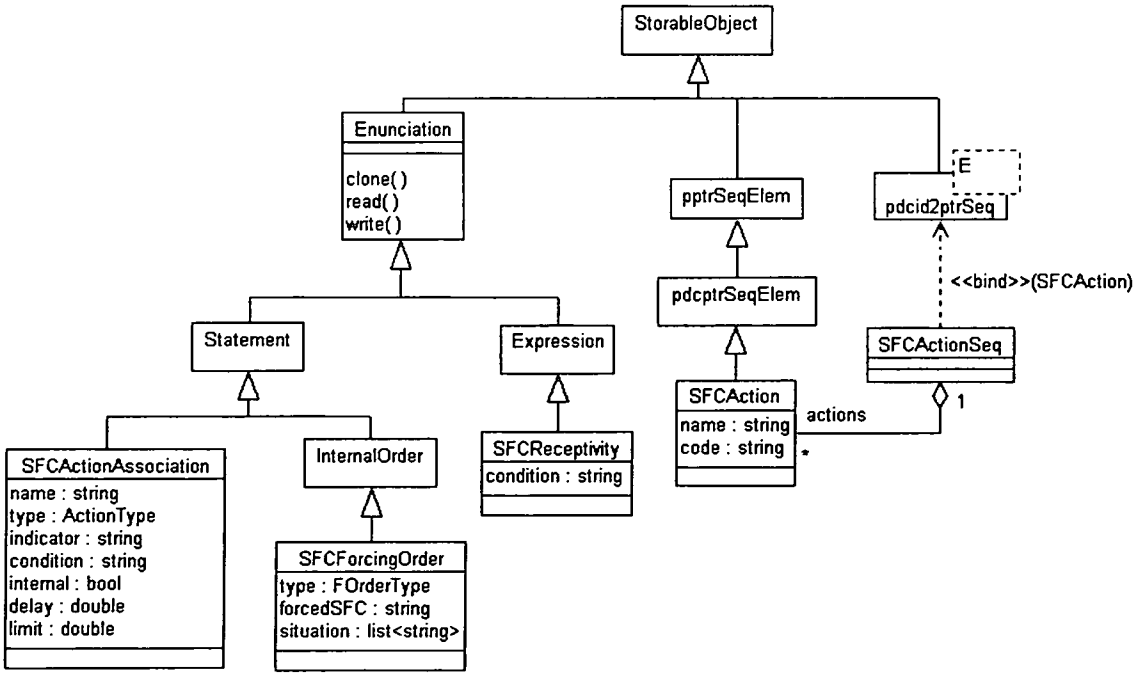


Figura 5.13. Diagrama das clases que implementan accións e receptividades.

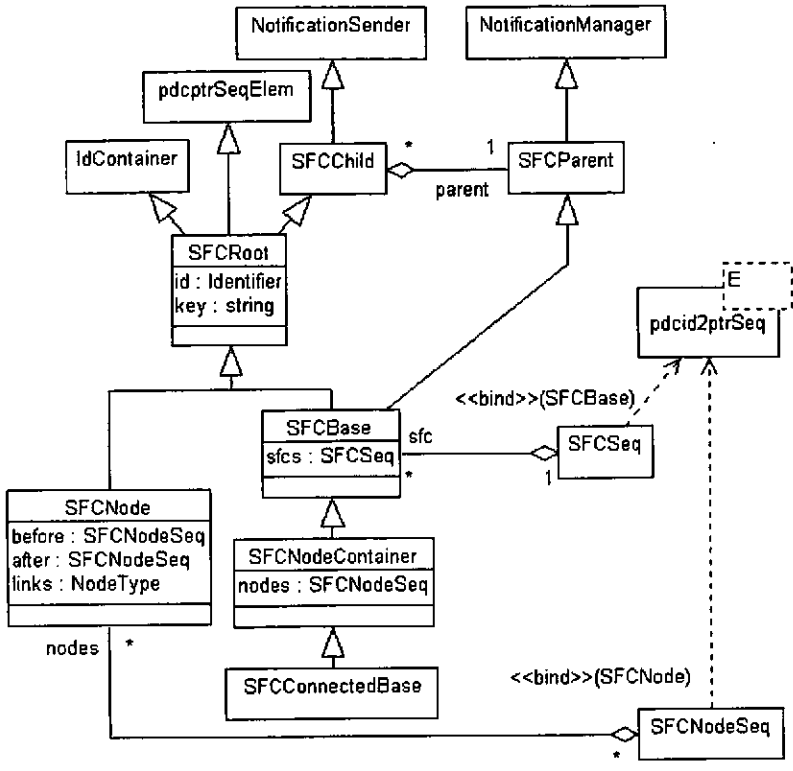


Figura 5.14. Diagrama das clases abstractas que dan soporte á estructura dos modelos Grafcet.

O diagrama de clases da Figura 5.15 mostra a especificación completa das clases que implementan as metaclases dos paquetes *Grafcet Core* e *Grafcet Hierarchy* do metamodelo e cal é a súa relación coas clases abstractas da Figura 5.14. As clases *SFCGlobal* e *SFCPartial* conteñen a información sobre a utilización de identificadores no modelo —nos atributos *pgids* e *nodeids*, respectivamente—. A clase *SFCGlobal* leva rexistro dos identificadores dos grafcets parciais que contén, e a clase *SFCParcial* os dos nodos[45]. Nótese que a clase *SFCMacro* especializa tanto á clase *SFCNode* como á clase *SFCNodeContainer*, polo que implementa simultaneamente a semántica dun nodo e a dun nivel da xerarquía estructural que contén outros nodos (os da súa macroexpansión).

A libraría complétase con tres clases que implementan diferentes algoritmos para a verificación de propiedades sintácticas. Estas clases son as seguintes:

- *SFCFlattener*, dado un modelo calcula o equivalente despois de substituír as macroetapas polas súas correspondentes macroexpansións. Polo tanto o resultado é o grafcet equivalente ao orixinal sen macroetapas.
- *SFCMacroChecker*, comproba a corrección sintáctica das macroexpansións dun modelo (§3.2.2.2).
- *SFCFOChecker*, comproba a coherencia da xerarquía de forzado dun modelo (§3.2.2.4).

[45] A libraría non implementa un mecanismo de identificación automático para os grafcets conexos, ainda que si garante a súa unicidade.

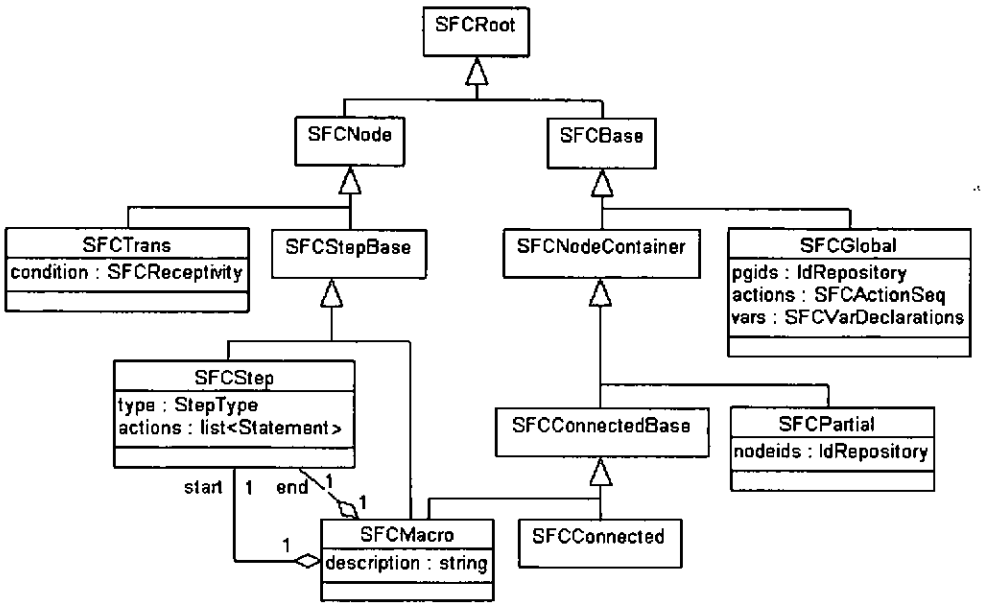


Figura 5.15. Diagrama das clases que implementan a estructura dos modelos Grafcet.

5.2.3. Implementación das operacións básicas

Neste apartado descríbense as notificacións identificadas durante a implementación e danse algúns detalles de como se implementou algunha das operacións básicas da libraría. A Táboa 5-I mostra as notificacións relacionadas con cambios na estructura dos modelos e a Táboa 5-II as relacionadas coa identificación dos elementos que conteñen. Para cada notificación indícanse as clases que a procesan e dáse unha breve descrición das operacións que realizan.

5.2.3.1. Inserción dun nodo

As operacións a realizar para a inserción dun nodo dependen do nivel da xerarquía do modelo no que se insira. Basicamente consisten en comprobar a existencia de duplicados no modelo (o que require un percorrido recursivo da estructura xerárquica), almacenar referencias ao novo nodo, reservar o seu identificador (información almacenada en cada grafcet parcial) e, se é o caso, crear un novo grafcet conexo que o conteña. Na Figura 5.16, na Figura 5.17 e na Figura 5.18 móstranse as secuencias de mensaxes que implementan a inserción dun nodo nun grafcet parcial, nun grafcet conexo e na macroexpansión dunha macroetapa, respectivamente.

5.2.3.2. Inserción dunha macroetapa

A inserción dunha macroetapa é semellante á dun nodo. A principal diferencia consiste en que a comprobación de duplicados, o almacenamento de referencias e a reserva de identificadores ten que ter en conta non só a macroetapa senón tamén os contidos da súa macroexpansión (que pode conter macroetapas aniñadas). A Figura 5.19 mostra a secuencia de mensaxes que implementa a inserción dunha macroetapa nun grafcet parcial.

5.2.3.3. Eliminación dun nodo

A operación de eliminación consiste, basicamente, na eliminación de referencias ao nodo, a liberación do seu identificador e a actualización da información dos grafcets conexos no caso de que, despois da eliminación do nodo, o grafcet conexo que o contiña quede dividido en un ou máis grafcets conexos. As macroetapas son eliminadas do mesmo xeito, coa diferencia de

que a eliminación de referencias e a liberación de identificadores ten que ter en conta tamén os contidos da súa macroexpansión. A Figura 5.20 mostra a secuencia de mensaxes que implementan a eliminación dun nodo dun grafcet parcial.

5.2.3.4. Eliminación dun grafcet conexo

A eliminación dun grafcet conexo é mais simple ca dun nodo, e consiste na eliminación de referencias ao grafcet conexo, a liberación do seu identificador e a eliminación de referencias e liberación dos identificadores dos nodos que contén. A Figura 5.21 mostra a secuencia de mensaxes que implementan a eliminación dun grafcet conexo contido nun grafcet parcial.

5.2.3.5. Modificación do identificador dun nodo

A modificación do identificador dun nodo require actualizar toda a información contida no modelo que faga referencia a ese nodo a través do seu identificador. Isto implica comprobar a existencia de duplicados co novo identificador, liberar o identificador vello, eliminar as referencias que utilicen o identificador vello tanto na estructura xerárquica como nos nodos conectados ao nodo modificado, reservar o novo identificador e actualizar as referencias ao nodo utilizando o novo identificador. A Figura 5.22 mostra a secuencia de mensaxes que implementan a modificación do identificador dun nodo contido na macroexpansión dunha macroetapa. Nótese que a operación de actualización de referencias ao nodo co novo identificador faise utilizando a operación descrita na Figura 5.18.

Notificación	Causa	Clase	Acción
INSERTCONTENTS_SFC	Inserción de novos nodos nun nivel da xerarquía estructural.	SFCPartial	Almacenar referencias (nodos)
DELETECONTENTS_SFC	Eliminación de nodos nun nivel da xerarquía estructural.	SFCPartial	Eliminar referencias (nodos)
DELETE_SFC	Destrucción dunha partición nun nivel da xerarquía estructural.	SFCGlobal	Eliminar referencia (grafcet parcial) Liberar id. (grafcet parcial)
		SFCPartial	Eliminar referencia (grafcet conexo) Liberar ids. (contidos grafcet conexo)
		SFCConnected	Notificar DELETE (grafcet parcial pai) Eliminar referencia (macroetapa) Liberar id. (macroetapa) Eliminar referencias nodos (macroexpansión) Liberar ids. nodos (macroexpansión) Calcular novos grafcets conexos Actualizar o grafcet parcial pai
		SFCMacro	Eliminar referencia (macroetapa) Liberar id. (macroetapa) Eliminar referencias nodos (macroexpansión) Liberar ids. nodos (macroexpansión)
ERASE_SFC	Eliminación dunha partición nun nivel da xerarquía estructural.	SFCGlobal	Idem DELETE_SFC
		SFCPartial	Idem DELETE_SFC
		SFCConnected	Notificar DELETE (grafcet parcial pai) Eliminar referencia (macroetapa) Liberar id. (macroetapa) Eliminar referencias nodos (macroexpansión) Liberar ids. nodos (macroexpansión)
		SFCMacro	Idem DELETE_SFC
INSERT_SFC	Inserción dunha nova partición nun nivel da xerarquía estructural.	SFCGlobal	Comprobar duplicados (id. grafcet parcial) Inserir referencia (grafcet parcial) Bloquear id. (grafcet parcial)
		SFCPartial	Comprobar duplicados (id. grafcet conexo) Comprobar duplicados (contidos grafcet conexo) Inserir referencia (grafcet conexo) Bloquear id. (grafcet conexo) Bloquear ids. (contidos grafcet conexo)
		SFCConnected	Inserir referencia (macroetapa) Bloquear id. (macroetapa) Inserir referencias nodos (macroexpansión) Bloquear ids. nodos (macroexpansión) Notificar INSERT (grafcet parcial pai)
		SFCMacro	Inserir referencia (macroetapa) Bloquear id. (macroetapa) Inserir referencias nodos (macroexpansión) Bloquear ids. nodos (macroexpansión)
DELETE	Destrucción dun nodo.	SFCPartial	Eliminar referencia (nodo)
		SFCConnected	Notificar DELETE (grafcet parcial pai) Eliminar nodo Liberar id. (nodo) Calcular novos grafcets conexos Actualizar o grafcet parcial pai
		SFCMacro	Notificar UNLOCKID Eliminar nodo
INSERT	Inserción dun novo nodo.	SFCPartial	Inserir referencia (nodo)
		SFCConnected	Notificar INSERT (grafcet parcial pai)

Táboa 5-I. Notificacións relacionadas coa modificación da estructura dun modelo Grafcet.

Notificación	Causa	Clase	Acción
SETID_SFC	Modificación do identificador dun nivel da xerarquía estructural.	SFCGlobal	Comprobar duplicados (novo id. grafcet parcial) Liberar id. vello Actualizar id. (grafcet parcial) Bloquear novo id.
		SFCPartial	Comprobar duplicados (novo id. grafcet conexo) Liberar id. vello Actualizar id. (grafcet conexo) Bloquear novo id.
		SFCConnected	Notificar DELETE (grafcet parcial pai) Comprobar duplicados (novo id. macroetapa) Liberar id. vello Actualizar id. (macroetapa) Bloquear novo id. Notificar INSERT (grafcet parcial pai)
		SFCMacro	Comprobar duplicados (novo id. macroetapa) Liberar id. vello Actualizar id. (macroetapa) Bloquear novo id.
LOCKID_SFC	Bloqueo do identificador dun nivel da xerarquía estructural.	SFCGlobal	Bloquear id. (grafcet parcial)
		SFCPartial	Bloquear id. (grafcet conexo)
		SFCConnected	Notificar LOCKID_SFC (grafcet parcial pai)
		SFCMacro	Notificar LOCKID_SFC
UNLOCKID_SFC	Liberación do identificador dun nivel da xerarquía estructural.	SFCGlobal	Liberar id. (grafcet parcial)
		SFCPartial	Liberar id. (grafcet conexo)
		SFCConnected	Notificar UNLOCKID_SFC (grafcet parcial pai)
		SFCMacro	Notificar UNLOCKID_SFC
SETID	Modificación do identificador dun nodo.	SFCConnected	Notificar DELETE (grafcet parcial pai) Comprobar duplicados (novo id. nodo) Liberar id. vello Actualizar id. (nodo) Bloquear novo id. Notificar INSERT (grafcet parcial pai)
		SFCMacro	Comprobar duplicados (novo id. nodo) Liberar id. vello Actualizar id. (nodo) Bloquear novo id.
CHECKID	Consulta do estado de bloqueo do identificador dun nodo.	SFCGlobal	Comprobar duplicados (id. grafcet parcial)
		SFCPartial	Comprobar duplicados (id. grafcet conexo) Comprobar duplicados (contidos grafcet conexo)
		SFCConnected	Notificar CHECKID (grafcet parcial pai)
		SFCMacro	Notificar CHECKID
LOCKID	Bloqueo do identificador dun nodo.	SFCPartial	Bloquear id. (nodo)
		SFCConnected	Notificar LOCKID (grafcet parcial pai)
		SFCMacro	Notificar LOCKID
UNLOCKID	Liberación do identificador dun nodo.	SFCPartial	Liberar id. (nodo)
		SFCConnected	Notificar UNLOCKID (grafcet parcial pai)
		SFCMacro	Notificar UNLOCKID

Táboa 5-II. Notificacións relacionadas coa identificación de elementos dun modelo Grafcet.

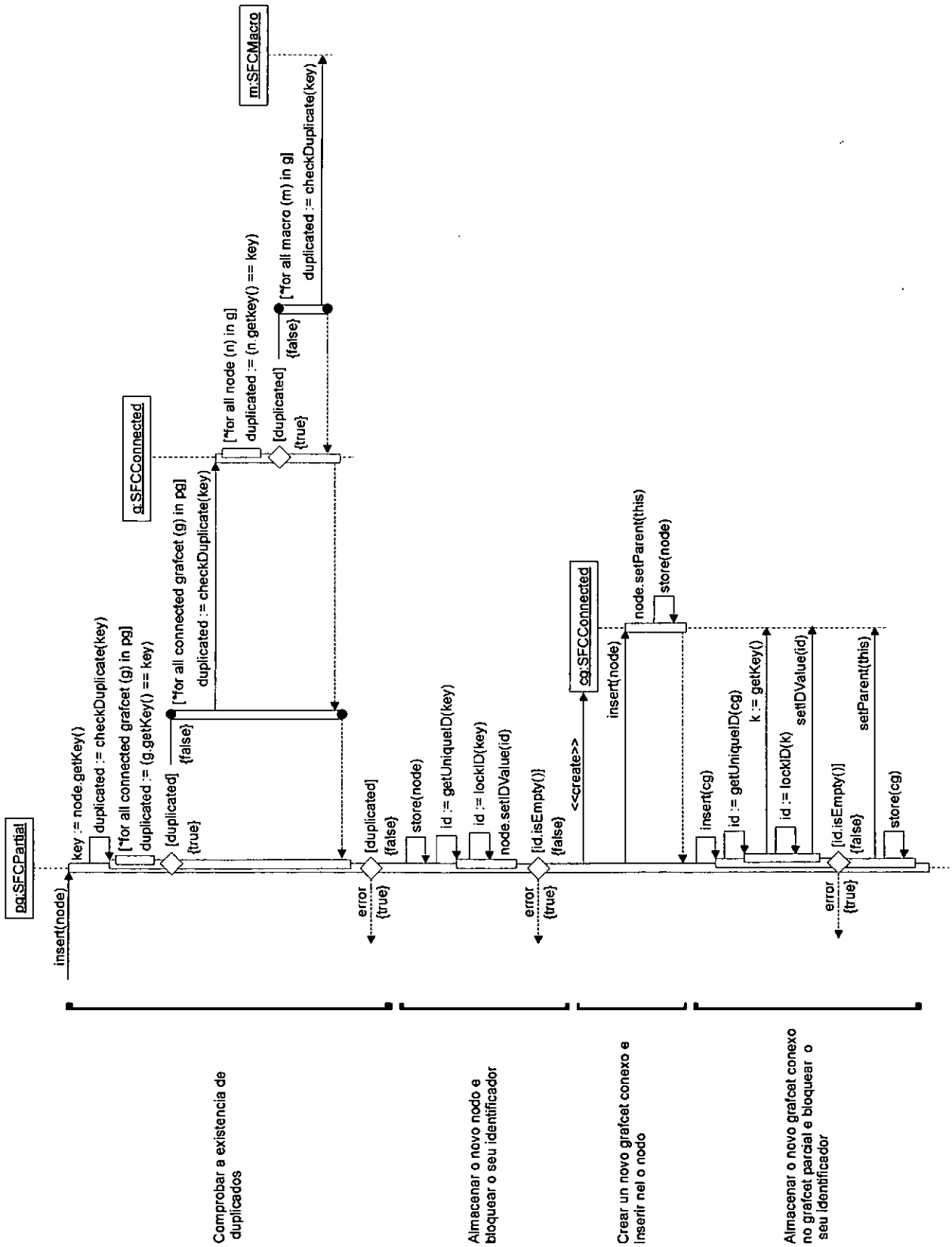


Figura 5.16. Secuencia de mensaxes da inserción dun nodo nun grafcet parcial.

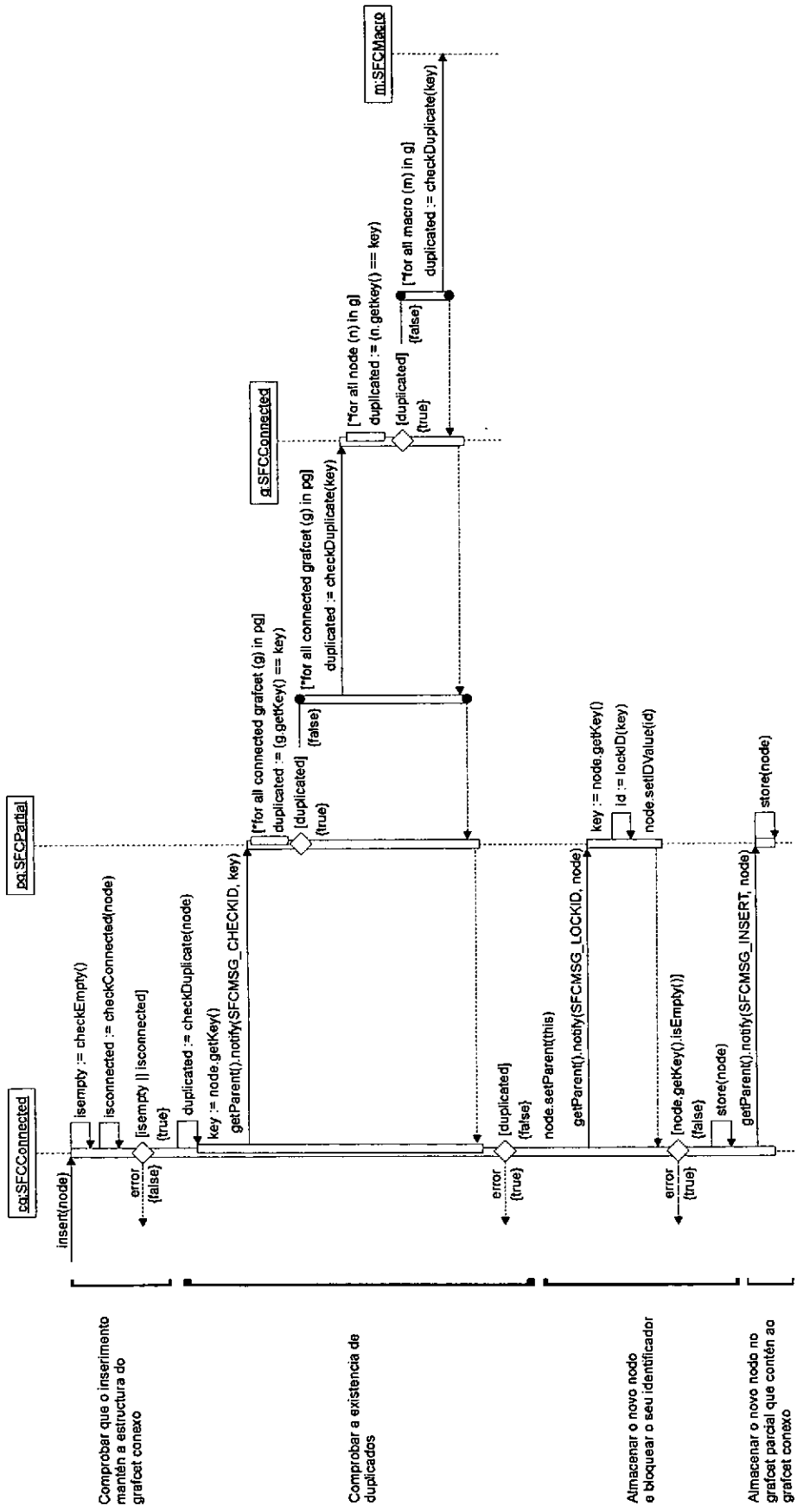


Figura 5.17. Secuencia de mensaxes da inserción dun nodo nun grafcet conexo.

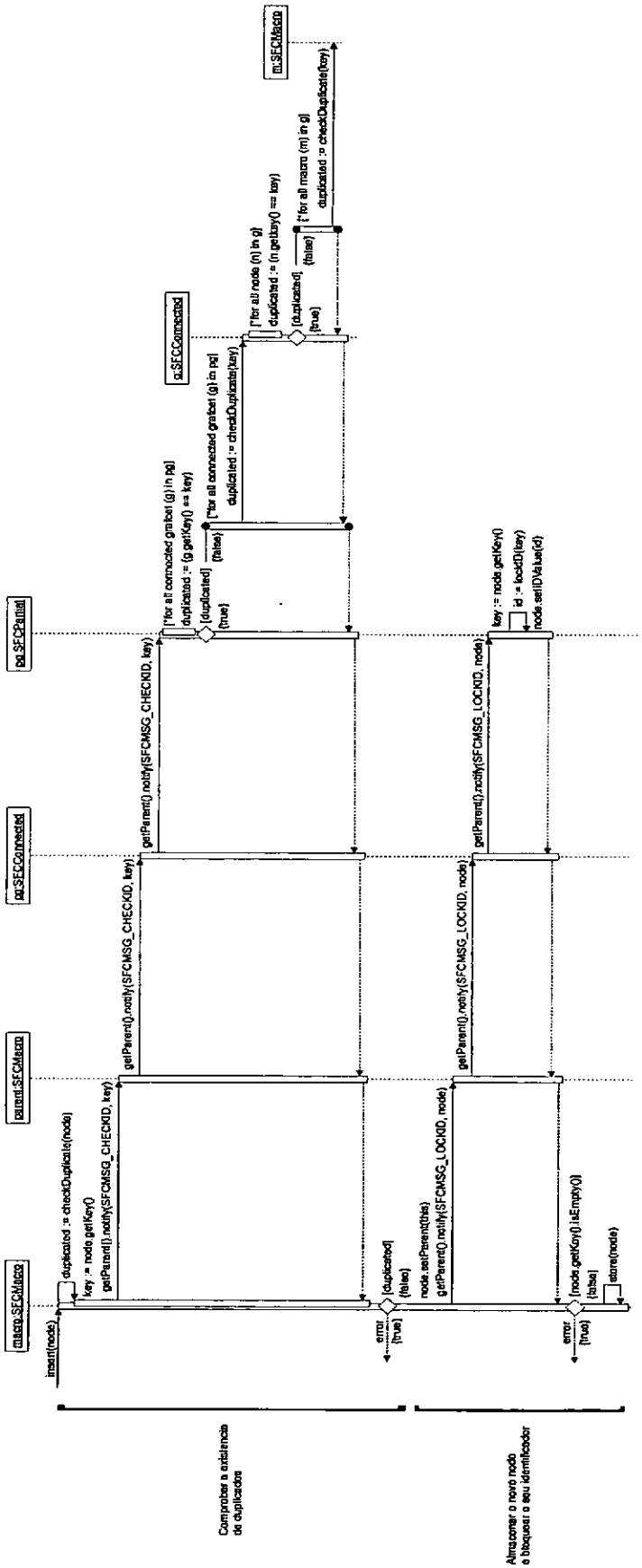


Figura 5.18. Secuencia de mensaxes da inserción dun nodo na macroexpansión dunha macroetapa.

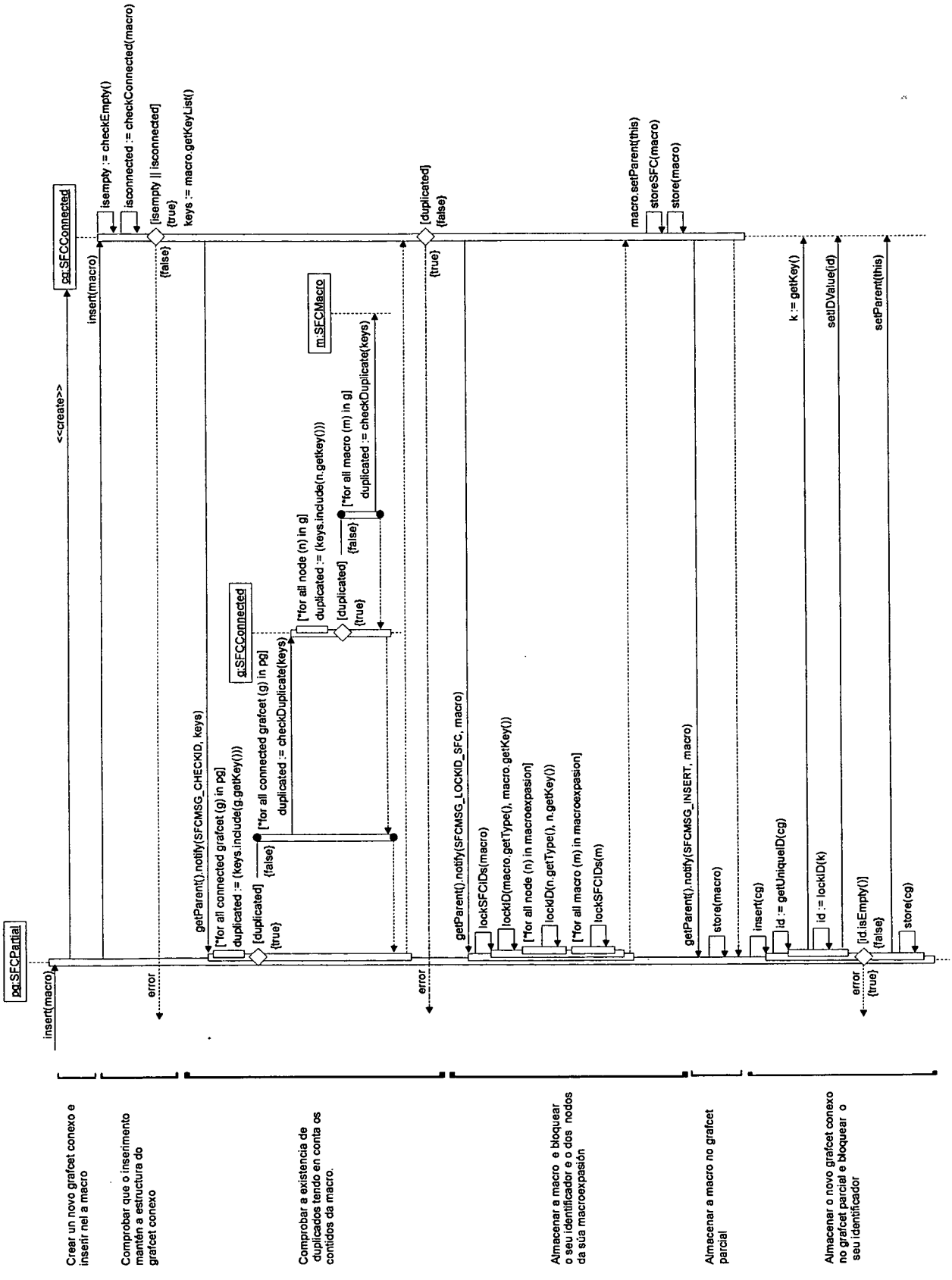


Figura 5.19. Secuencia de mensaxes da inserción dunha macroetapa nun grafcet parcial.

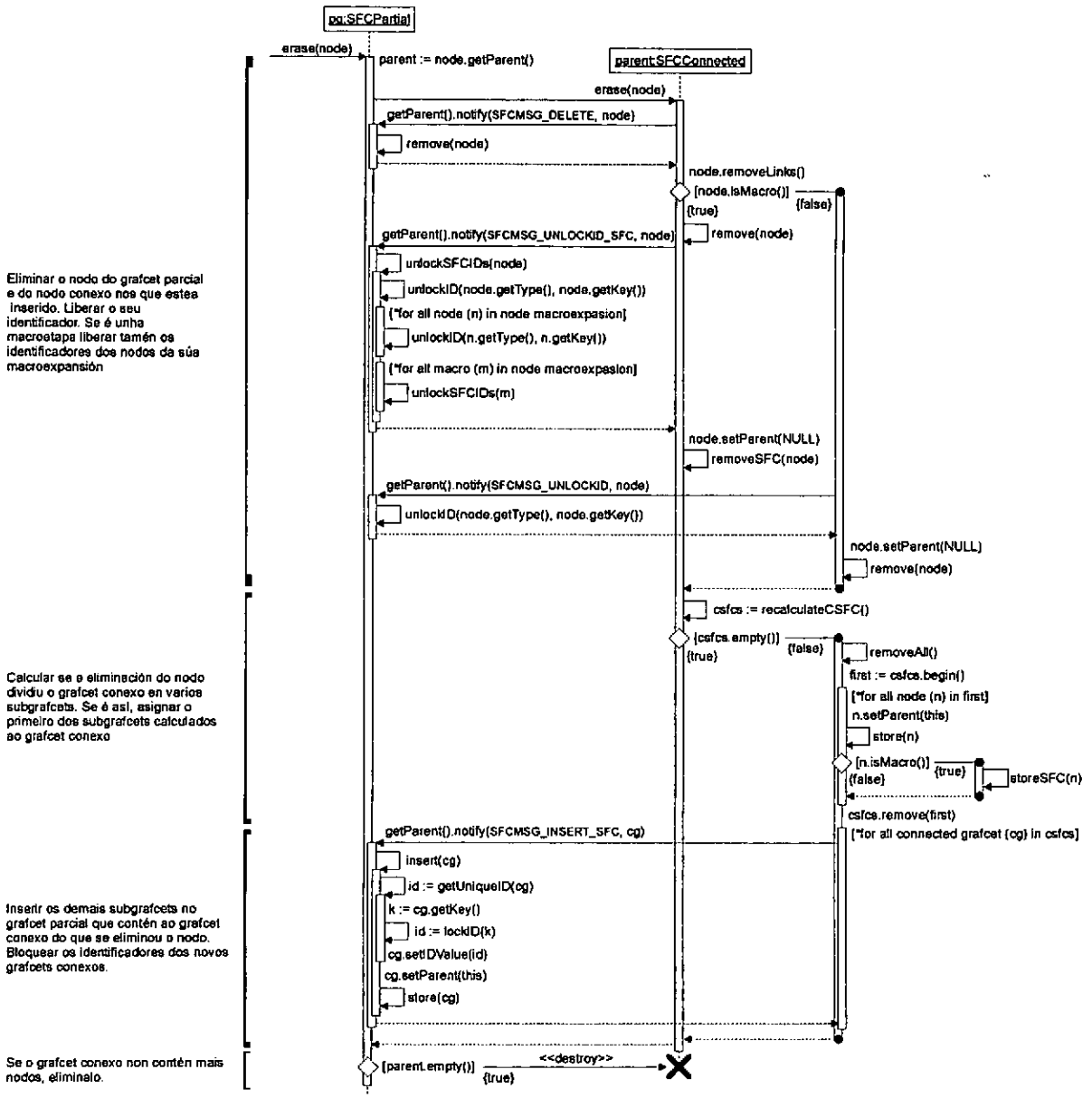


Figura 5.20. Secuencia de mensaxes da eliminación dun nodo do modelo.

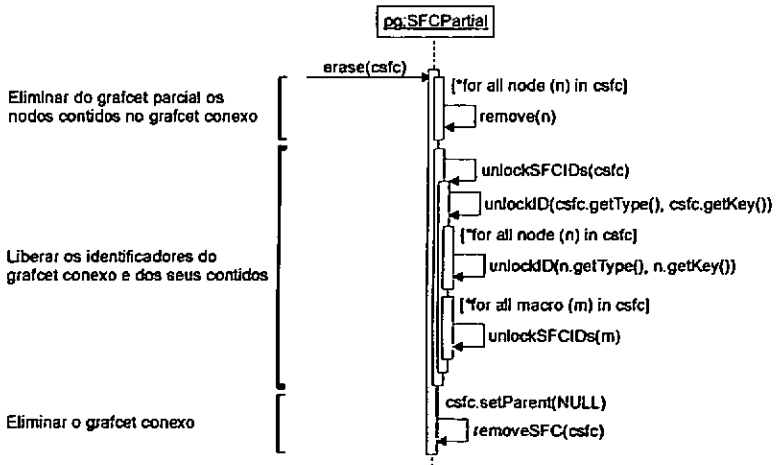


Figura 5.21. Secuencia de mensaxes da eliminación dun grafcet conexo do modelo.

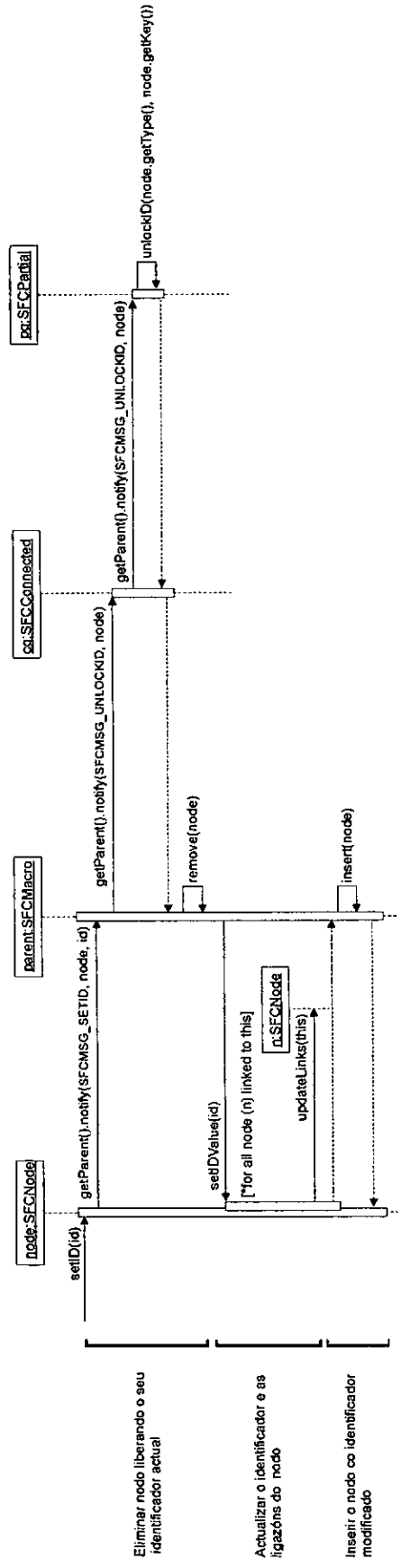


Figura 5.22. Secuencia de mensaxes da modificación do identificador dun nodo incluído nunha macroexpansión.

5.3. Exemplos de modelado

Para ilustrar a utilización do metamodelo e da libraría que o implementa, neste apartado danse varios exemplos da súa utilización. Cada exemplo mostra unha estructura Grafcet, o diagrama de obxectos que a representa mediante os conceptos definidos no metamodelo proposto, e o código C++ que crea a estructura en memoria utilizando a libraría implementada.

5.3.1. Estructuras básicas

5.3.1.1. Secuencia

```
19. // tipos de datos: instanciacións explícitas
20. template GescaVarDecl<bool>;
21. template GescaVar<bool>;
22. typedef GescaVarDecl<bool> SFC_bool;
23. template GescaSystemVarDecl<bool>;
24. template GescaSystemVar<bool>;
25. typedef GescaSystemVarDecl<bool> SFC_system_bool;
26. SystemDataDeclSeq process_vars;
27. // creación das entradas booleanas e asignación aos canais de E/S
28. SFC_system_bool* input1 = new SFC_system_bool("a",
29.                     "SIM:TCP_BOOL_8I_8O",
30.                     "\\\\sim:tcp_bool_8i_8o\\sim:tcp_bool_8i_8o\\in_channel\\in_channel_1",
31.                     READ);
32. process_vars.insert(input1);
33. // modelo Grafcet
34. SFCGlobal global("GG");
35. // variábeis do modelo
36. SFC_bool* var1 = new SFC_bool("A");
37. global.vars.insert(var1);
38. // grafcets parciais
39. SFCPartial* sfc1 = new SFCPartial("PG1");
40. global.insert(sfc1);
41. // etapas
42. SFCStep step0("0", SFCStep::initial);
43. SFCStep step1("1");
44. SFCStep step2("2");
45. // asociacións etapa-acción
46. SFCActionAssociation* association1 = new SFCActionAssociation("A", S);
47. SFCActionAssociation* association2 = new SFCActionAssociation("A",
48.                                                     R, "", "", 0, 0, true);
49. step1.addAction(association1);
50. step2.addAction(association2);
51. // transicións
52. SFCTrans trans0("(0)");
53. SFCTrans trans1("(1)");
54. SFCTrans trans2("(2)");
55. // condicións de transición
56. trans0.m_condition.setCondition("↑a");
57. trans1.m_condition.setCondition("↑a");
58. trans2.m_condition.setCondition("a");
59. // enlaces directos
60. node_iterator s0 = sfc1->insert(step0);
61. node_iterator t0 = sfc1->insertAfter(s0, trans0);
62. node_iterator s1 = sfc1->insertAfter(t0, step1);
63. node_iterator t1 = sfc1->insertAfter(s1, trans1);
64. node_iterator s2 = sfc1->insertAfter(t1, step2);
65. node_iterator t2 = sfc1->insertAfter(s2, trans2);
66. sfc1->link(t2, s0);
```

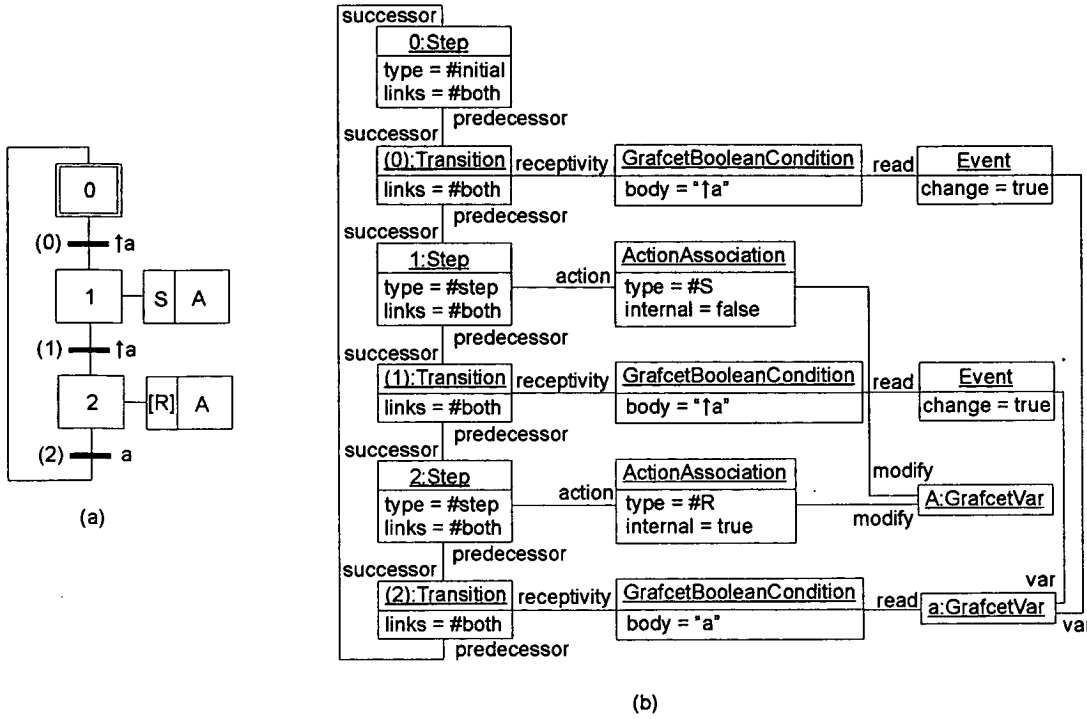


Figura 5.23. Exemplo de: a) secuencia; e b) representación co metamodelo proposto.

5.3.1.2. Selección de secuencia

```
67. // modelo Grafcet
68. SFCGlobal global("GG");
69. // grafcets parciais
70. SFCPartial* sfc1 = new SFCPartial("PG1");
71. global.insert(sfc1);
72. // etapas
73. SFCStep step0("0", SFCStep::initial, SFCStep::out);
74. SFCStep step1("1");
75. SFCStep step2("2");
76. SFCStep step3("3");
77. SFCStep step4("4");
78. // transicións
79. SFCTrans trans0a("(0.a)");
80. SFCTrans trans0b("(0.b)");
81. SFCTrans trans1a("(1.a)");
82. SFCTrans trans1b("(1.b)");
83. SFCTrans trans2("(2)");
84. SFCTrans trans3("(3)");
85. SFCTrans trans4("(4)", "", SFCTrans::in);
86. // enlaces directos
87. node_iterator s0 = sfc1->insert(step0);
88. node_iterator t0a = sfc1->insertAfter(s0, trans0a);
89. node_iterator t0b = sfc1->insertAfter(s0, trans0b);
90. node_iterator s1 = sfc1->insertAfter(t0b, step1);
91. node_iterator t1a = sfc1->insertAfter(s1, trans1a);
92. node_iterator t1b = sfc1->insertAfter(s1, trans1b);
93. node_iterator s2 = sfc1->insertAfter(t1a, step2);
94. node_iterator t2 = sfc1->insertAfter(s2, trans2);
95. node_iterator s3 = sfc1->insertAfter(t1b, step3);
96. node_iterator t3 = sfc1->insertAfter(s3, trans3);
97. node_iterator s4 = sfc1->insertAfter(t0a, step4);
98. sfc1->link(t2, s4);
99. sfc1->link(t3, s4);
100. node_iterator t4 = sfc1->insertAfter(s4, trans4);
```

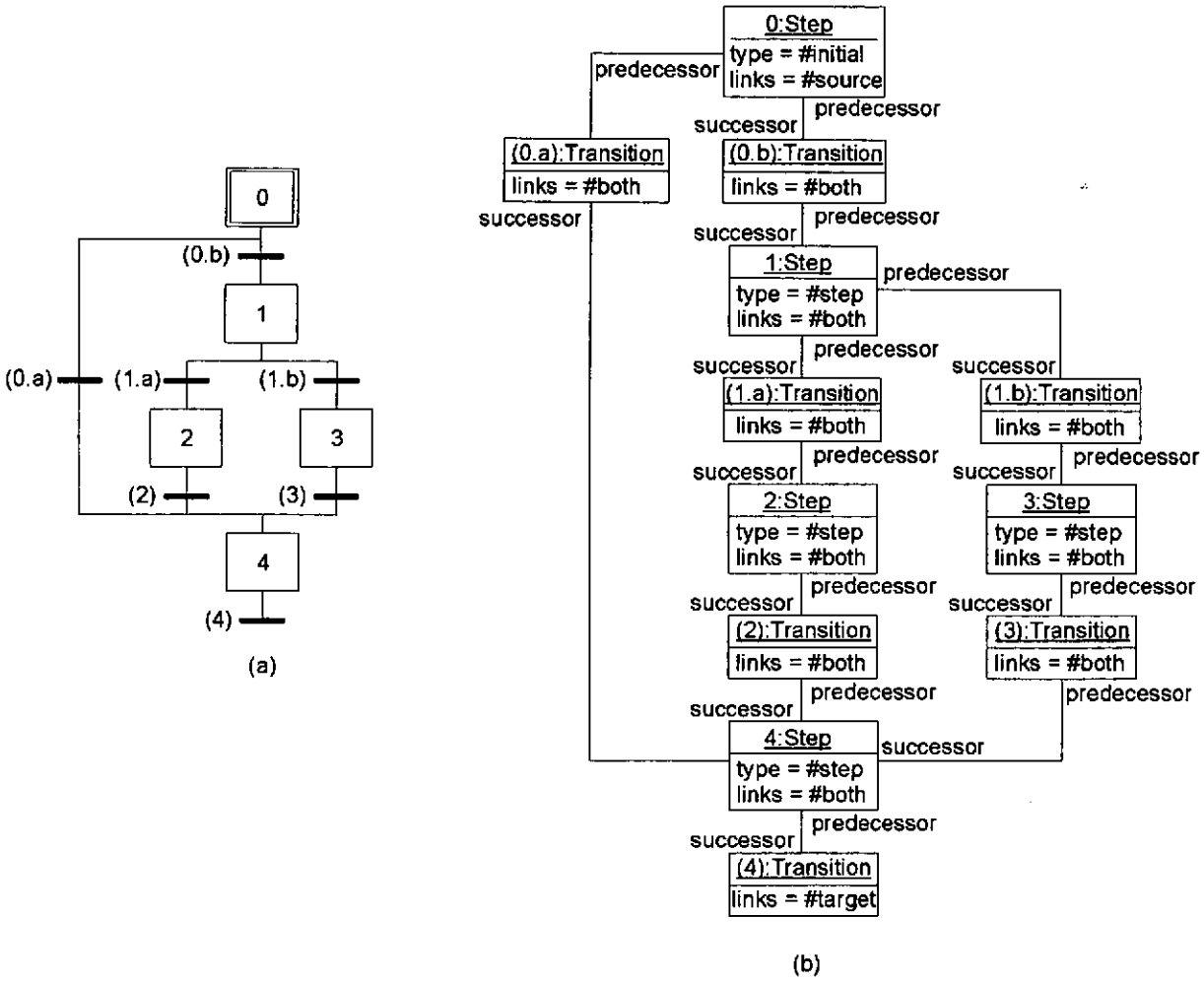


Figura 5.24. Exemplo de: a) selección de secuencia; e b) representación co metamodelo proposto.

5.3.1.3. Paralelismo

```
101. // modelo Grafcet
102. SFCGlobal global("GG");
103. // grafcets parciais
104. SFCPartial* sfc1 = new SFCPartial("PG1");
105. global.insert(sfc1);
106. // etapas
107. SFCStep step1("1", SFCStep::initial);
108. SFCStep step2("2");
109. SFCStep step3("3");
110. // transicións
111. SFCTrans trans0("(0)", "", SFCTrans::out);
112. SFCTrans trans1("(1)");
113. SFCTrans trans2("(2)");
114. // enlaces directos
115. node_iterator t0 = sfc1->insert(trans0);
116. node_iterator s1 = sfc1->insertAfter(t0, step1);
117. node_iterator t1 = sfc1->insertAfter(s1, trans1);
118. node_iterator s2 = sfc1->insertAfter(t1, step2);
119. node_iterator s3 = sfc1->insertAfter(t1, step3);
120. node_iterator t2 = sfc1->insertAfter(s2, trans2);
121. sfc1->link(s3, t2);
122. sfc1->link(t2, s1);
```

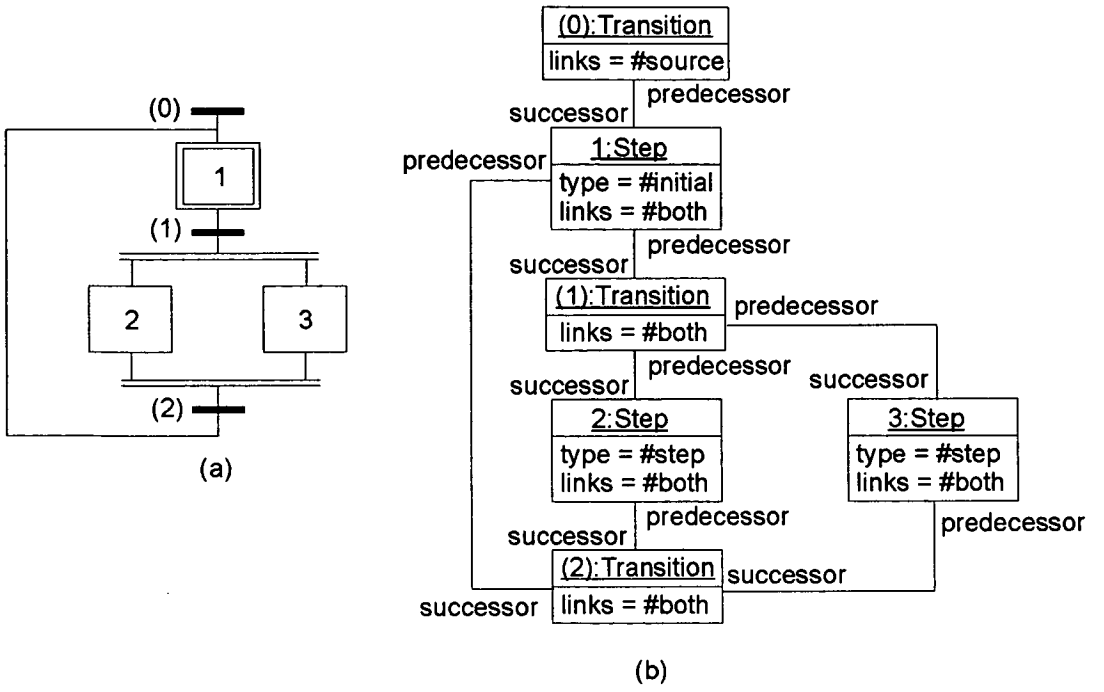


Figura 5.25. Exemplo de: a) secuencias paralelas; e b) representación co metamodelo proposto.

5.3.1.4. Semáforo

```
123. // modelo Grafcet
124. SFCGlobal global("GG");
125. // grafcets parciais
126. SFCPartial* sfc1 = new SFCPartial("PG1");
127. global.insert(sfc1);
128. // etapas
129. SFCStep step1("1", SFCStep::initial);
130. SFCStep step2("2");
131. SFCStep step3("3", SFCStep::initial);
132. SFCStep step4("4", SFCStep::initial);
133. SFCStep step5("5");
134. // transicións
135. SFCTrans trans1("(1)");
136. SFCTrans trans2("(2)");
137. SFCTrans trans3("(3)");
138. SFCTrans trans4("(4)");
139. // enlaces directos
140. node_iterator s1 = sfc1->insert(step1);
141. node_iterator t1 = sfc1->insertAfter(s1, trans1);
142. node_iterator s2 = sfc1->insertAfter(t1, step2);
143. node_iterator t2 = sfc1->insertAfter(s2, trans2);
144. node_iterator s3 = sfc1->insertBefore(t2, step3);
145. sfc1->link(t2, s1);
146. sfc1->link(t2, s3);
147. node_iterator s4 = sfc1->insert(step4);
148. node_iterator t4 = sfc1->insertAfter(s4, trans4);
149. node_iterator s5 = sfc1->insertAfter(t4, step5);
150. node_iterator t3 = sfc1->insertAfter(s5, trans3);
151. sfc1->link(s3, t3);
152. sfc1->link(t3, s3);
153. sfc1->link(t3, s4);
```

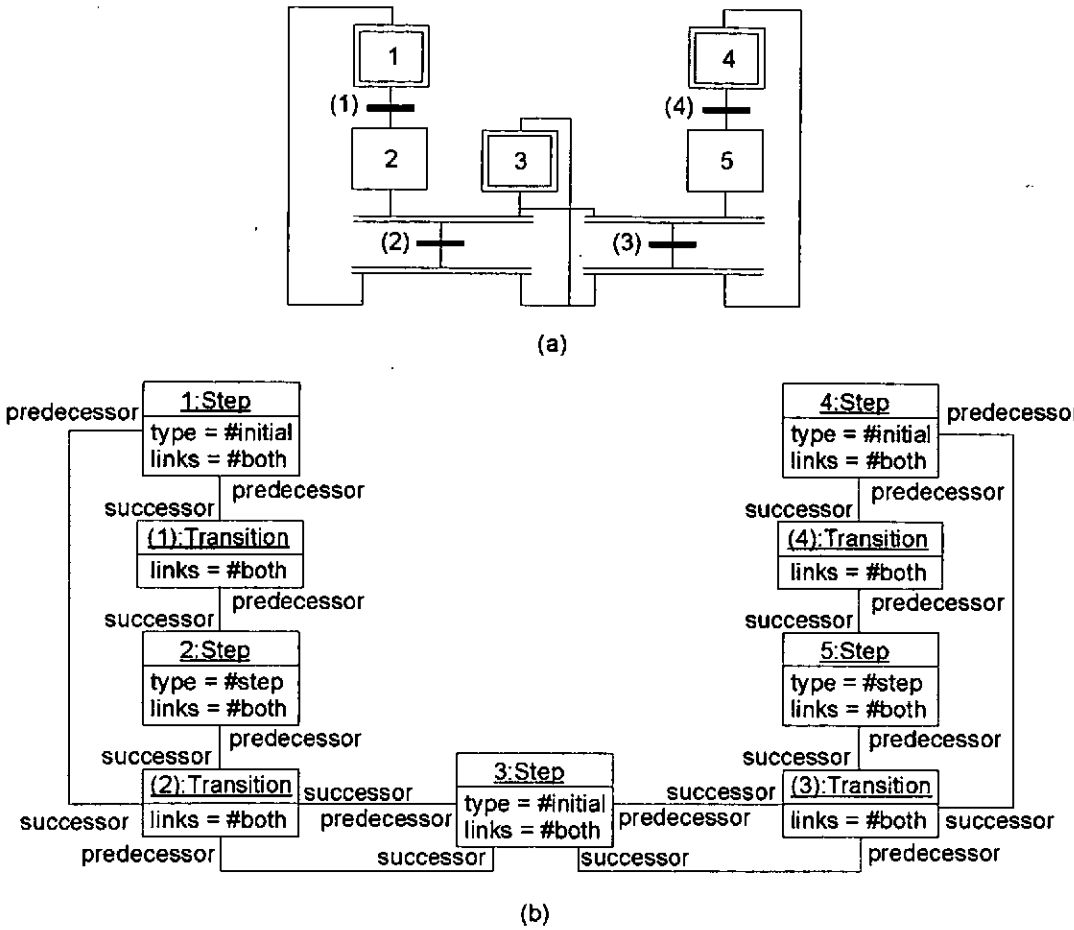


Figura 5.26. Exemplo de: a) secuencias exclusivas (semáforo); e b) representación co metamodelo proposto.

5.3.1.5. Accións

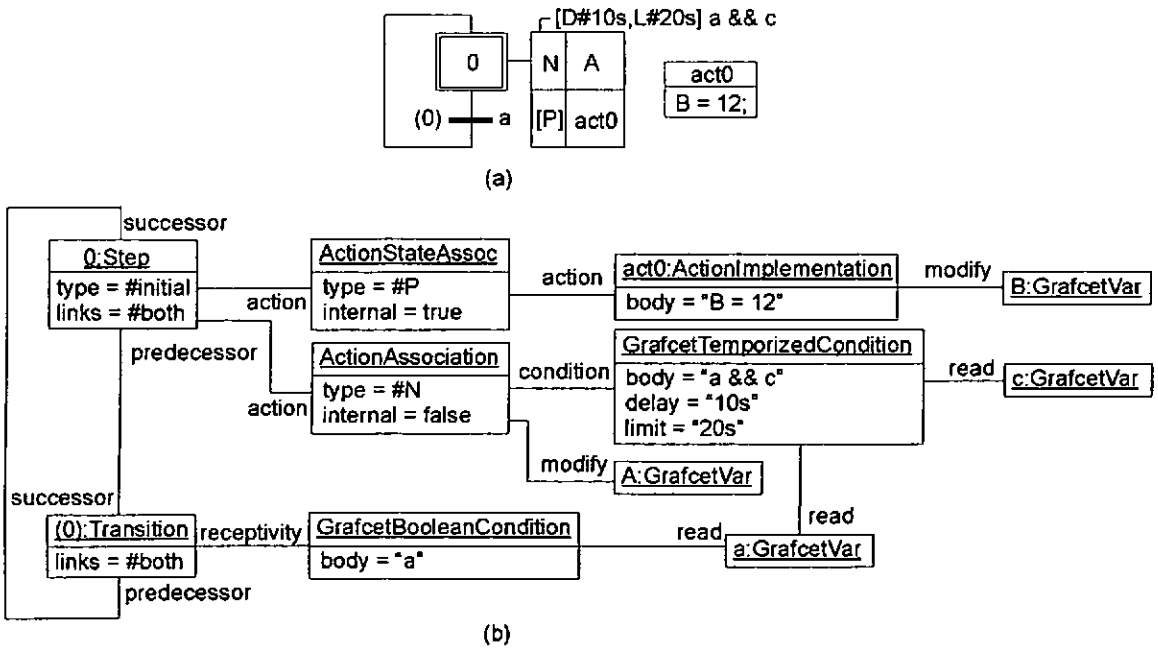


Figura 5.27. Exemplo de: a) accións; e b) representación co metamodelo proposto.

```
154. // tipos de datos: instanciacións explícitas
155. template GescaVarDecl<bool>;
156. template GescaVar<bool>;
157. typedef GescaVarDecl<bool> SFC_bool;
158. template GescaSystemVarDecl<bool>;
159. template GescaSystemVar<bool>;
160. typedef GescaSystemVarDecl<bool> SFC_system_bool;
161. SystemDataDeclSeq process_vars;
162. // creación das entradas booleanas e asignación aos canais de E/S
163. SFC_system_bool* input1 = new SFC_system_bool("a",
164.              "DDSIM:TCP_BOOL_8I_8O",
165.              "\\\\DDSIM:TCP_BOOL_8I_8O\\DDSIM:TCP_BOOL_8I_8O\\INPUT_CHANNELS\\INPUT_CHANNEL_1",
166.              READ);
167. SFC_system_bool* input2 = new SFC_system_bool("c",
168.              "DDSIM:TCP_BOOL_8I_8O",
169.              "\\\\DDSIM:TCP_BOOL_8I_8O\\DDSIM:TCP_BOOL_8I_8O\\INPUT_CHANNELS\\INPUT_CHANNEL_2",
170.              READ);
171. process_vars.insert(input1);
172. process_vars.insert(input2);
173. // modelo Grafcet
174. SFCGlobal global("GG");
175. // variábeis do modelo
176. SFC_bool* var1 = new SFC_bool("A");
177. SFC_bool* var2 = new SFC_bool("B");
178. global.vars.insert(var1);
179. global.vars.insert(var2);
180. // bloques de código
181. SFCAction* action0 = new SFCAction("act0", "B=12;");
182. global.actions.insert(action0);
183. // grafcets parciais
184. SFCPartial* sfc1 = new SFCPartial("PG1");
185. global.insert(sfc1);
186. // etapas
187. SFCStep step0("0", SFCStep::initial);
188. // asociacións etapa-acción
189. SFCActionAssociation* association1 = new SFCActionAssociation("A",
190.                                                    N, "","a&&c", 10, 20);
191. SFCActionAssociation* association2 = new SFCActionAssociation("act0",
192.                                                    P, "","", 0, 0, true);
193. step0.addAction(association1);
194. step0.addAction(association2);
195. // transicións
196. SFCTrans trans0("(0)");
197. // condicións de transición
198. trans0.m_condition.setCondition("a");
199. // enlaces directos
200. node_iterator s0 = sfc1->insert(step0);
201. node_iterator t0 = sfc1->insertAfter(s0, trans0);
202. sfc1->link(t0, s0);
```

5.3.2. Xerarquía

5.3.2.1. Macroetapas

```
203. // modelo Grafcet
204. SFCGlobal global("GG");
205. // grafcets parciais
206. SFCPartial* sfc1 = new SFCPartial("PG1");
207. global.insert(sfc1);
208. // etapas
209. SFCStep step0("0", SFCStep::initial);
210. SFCStep step2("2");
```

```
211. // transicións
212. SFCTrans trans0("(0)");
213. SFCTrans trans1("(1)");
214. SFCTrans trans2("(2)");
215. // macroetapas
216. SFCMacro macro100("M100", "Macro expansion");
217. macro100.getStart()->setId("S100");
218. macro100.getEnd()->setId("E100");
219. // nodos da macrotapa
220. SFCStep step101("101");
221. SFCTrans trans100("100");
222. SFCTrans trans101("101");
223. // enlaces directos da macroetapa
224. node_iterator t100 = macro100.insertAfter(macro100.getStart(), trans100);
225. node_iterator s101 = macro100.insertAfter(t100, step101);
226. node_iterator t101 = macro100.insertAfter(s101, trans101);
227. macro100.link(t101, macro10.getEnd());
228. // enlaces directos
229. node_iterator s0 = sfc1->insert(step0);
230. node_iterator t0 = sfc1->insertAfter(s0, trans0);
231. node_iterator m100 = sfc1->insertAfter(t0, macro100);
232. node_iterator t1 = sfc1->insertAfter(m100, trans1);
233. node_iterator s2 = sfc1->insertAfter(t1, step2);
234. node_iterator t2 = sfc1->insertAfter(s2, trans2);
235. sfc1->link(t2, s0);
```

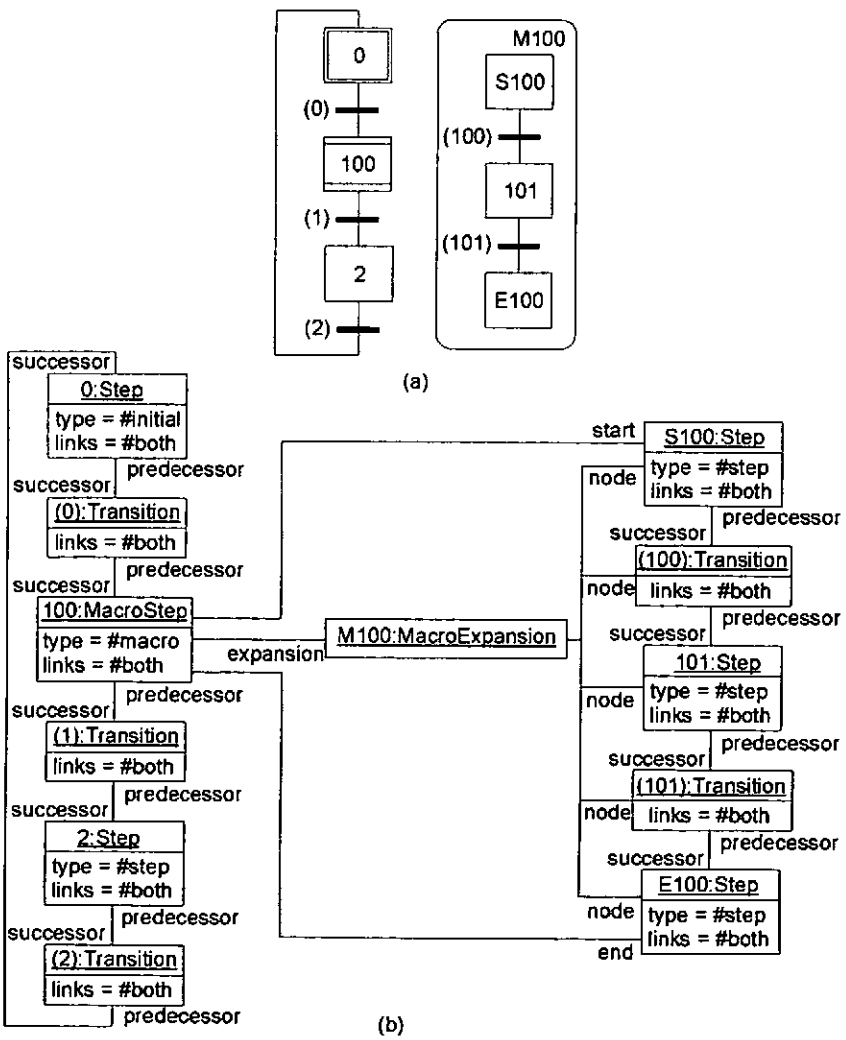


Figura 5.28. Exemplo de: a) macroetapa e macroexpansión; e b) representación co metamodelo proposto.

5.3.2.2. Ordes de forzado

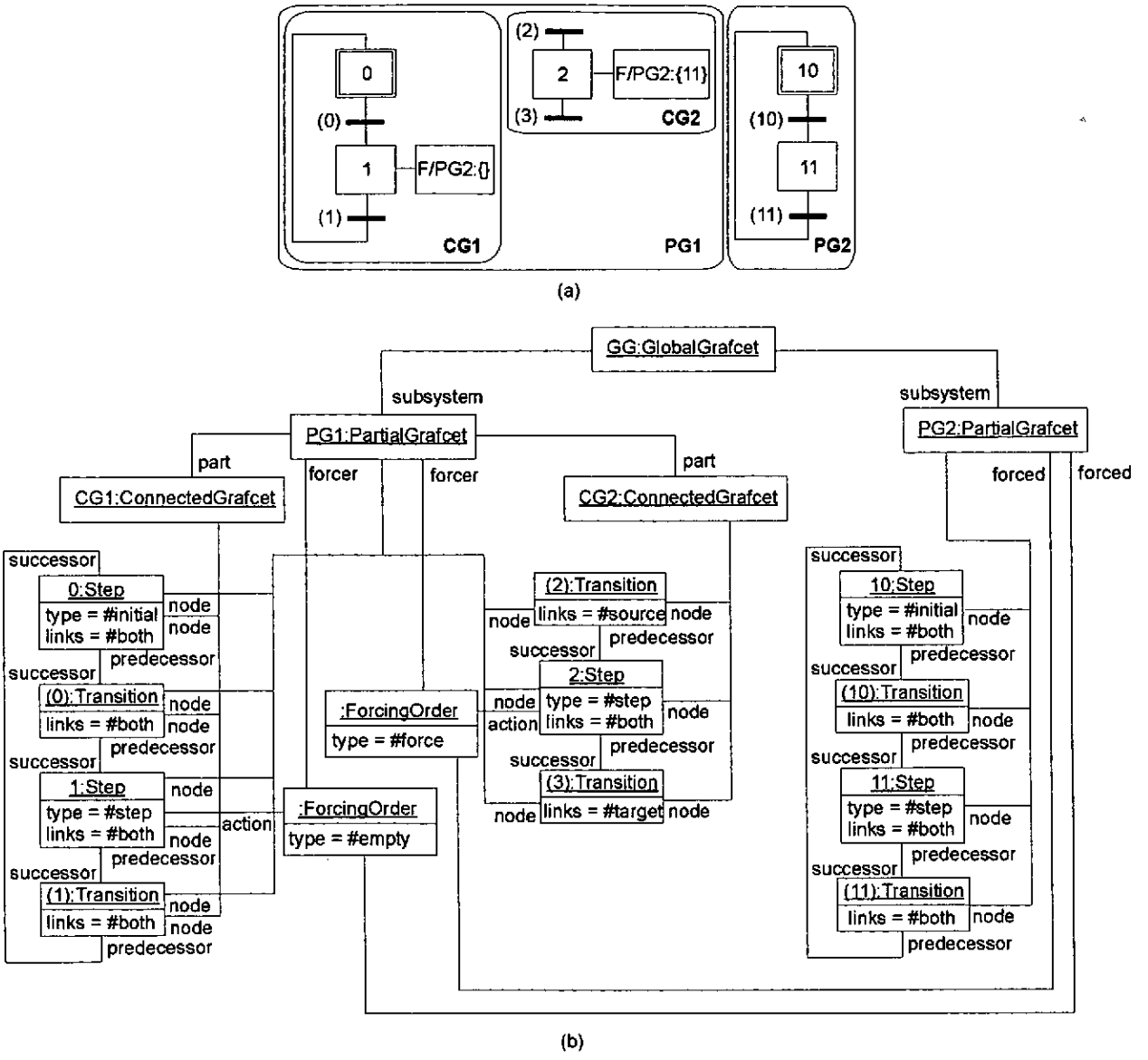


Figura 5.29. Exemplo de: a) ordes de forzado; e b) representación co metamodelo proposto[46].

```
236. // modelo Grafcet
237. SFCGlobal global("GG");
238. // grafcets parciais
239. SFCPartial* sfc1 = new SFCPartial("PG1");
240. SFCPartial* sfc2 = new SFCPartial("PG2");
241. global.insert(sfc1);
242. global.insert(sfc2);
243. // grafcets conexos
244. SFCConnected* csfc1 = new SFCConnected("CG1");
245. SFCConnected* csfc2 = new SFCConnected("CG2");
```

[46] Por razóns de simplicidade non se incluiu na figura a representación das situacións forzadas polas ordes de forzado.

```
246. // etapas
247. SFCStep step0("0", SFCStep::initial);
248. SFCStep step1("1");
249. SFCStep step2("2");
250. SFCStep step10("10", SFCStep::initial);
251. SFCStep step11("11");
252. // transicións
253. SFCTrans trans0("(0)");
254. SFCTrans trans1("(1)");
255. SFCTrans trans2("(2)", "", SFCTrans::out);
256. SFCTrans trans3("(3)", "", SFCTrans::in);
257. SFCTrans trans10("(10)");
258. SFCTrans trans11("(11)");
259. // ordes de forzado
260. SFCForcingOrder* forder1 = new SFCForcingOrder("PG2", FORCE_EMPTY);
261. step1.addAction(forder1);
262. list<string> situation;
263. situation.push_back("11");
264. SFCForcingOrder* forder2 = new SFCForcingOrder("PG2", situation);
265. step2.addAction(forder2);
266. // enlaces directos
267. node_iterator s0 = csfc1->insert(step0);
268. node_iterator t0 = csfc1->insertAfter(s0, trans0);
269. node_iterator s1 = csfc1->insertAfter(t0, step1);
270. node_iterator t1 = csfc1->insertAfter(s1, trans1);
271. csfc1->link(t1, s0);
272. node_iterator t2 = csfc1->insert(trans2);
273. node_iterator s2 = csfc1->insertAfter(t2, step2);
274. node_iterator t3 = csfc1->insertAfter(s2, trans3);
275. sfc1->insert(csfc1);
276. sfc1->insert(csfc2);
277. node_iterator s10 = sfc2->insert(step10);
278. node_iterator t10 = sfc2->insertAfter(s10, trans10);
279. node_iterator s11 = sfc2->insertAfter(t10, step11);
280. node_iterator t11 = sfc2->insertAfter(s11, trans11);
281. sfc2->link(t11, s10);
```

5.4. Conclusións

Neste capítulo propúxose un metamodelo para o Grafcet que define formalmente os conceptos e regras utilizados na creación de modelos coa ferramenta proposta nesta tese de doutoramento. Este metamodelo está integrado no de UML, polo que proporciona un medio de integración con outros formalismos especificados da mesma maneira (como acontece cos StateCharts), e facilita a utilización do Grafcet para a especificación de dinámicas complexas como parte de metodoloxías de desenvolvemento "software" baseadas en UML. Ademais completouse o metamodelo definindo un conxunto de invariantes na forma de regras OCL que poden ser utilizadas, mediante as ferramentas apropiadas, para a validación automática da correción sintáctica dos modelos Grafcet representados utilizando o metamodelo proposto.

A utilización práctica do metamodelo realízase mediante unha libraría C++, que implementa os conceptos e regras nel definidos, ademais dun conxuto de clases auxiliares para a comprobación da correción de características como a coherencia da xerarquía de forzado ou a estructura interna das macroexpansións. Esta libraría é un medio de integración do Grafcet en calqueira aplicación e, ademais, o soporte que inclúe á 'persistencia' dos modelos proporciona indirectamente un formato de almacenamento que pode ser utilizado para o intercambio destes entre aplicacións. As explicacións sobre a implementación da libraría non describen esta en detalle, limitándose a mostrar como son aplicados algún dos mecanismos e técnicas utilizados na implementación das operacións máis relevantes. O capítulo complétase con algúns exemplos de modelado de distintos aspectos sintácticos dos modelos Grafcet.

Capítulo 6. Compilación de modelos Grafcet

Para poder executar os modelos Grafcet representados mediante o metamodelo explicado en (§5.1), nos que o código de accións e receptividades é especificado utilizando unha linguaxe de programación de alto nivel (neste caso C++), é preciso realizar previamente a compilación do modelo para obter o código obxecto que será cargado e executado na máquina virtual. A compilación dun modelo Grafcet implica comprobar a súa corrección, crear unha representación optimizada para a súa execución, transformar o código de accións e receptividades para facelas compilábeis cun compilador C++, xerar o código fonte da DLL que será utilizada para cargar e executar o modelo na máquina virtual e, finalmente, compilalo para obter o código obxecto da DLL.

Neste capítulo descríbese o compilador Grafcet desenvolvido para realizar as operacións anteriores. No deseño da arquitectura lóxica do compilador buscouse unha estructura de fases flexíbel, que permitira en futuras versións engadir, eliminar ou modificar fases facilmente. Tamén se deseñou unha técnica que permite utilizar aplicacións externas durante a compilación para realizar diferentes operacións como, por exemplo, a simplificación de expresións lóxicas. Esta técnica permite cambiar a aplicación externa utilizada sen que sexa necesario modificar o código do compilador. Un aspecto do que o compilador é fortemente dependente é a linguaxe utilizada para a especificación de accións e receptividades. Nesta primeira versión, a implementación de varias das fases do compilador presupón que a linguaxe utilizada é o C++. En futuras versións podería incluírse soporte a outras linguaxes.

O resto do capítulo estructurouse do xeito seguinte: no apartado (§6.1) descríbese o proceso de compilación e os arquivos e aplicacións implicados; no apartado (§6.2) explícase a estructura de fases do compilador e a información interna que manexa durante a compilación; no apartado (§6.3) detállanse os aspectos relacionados coa utilización da linguaxe C++ para a especificación do código de accións e receptividades, así como as extensións que foi preciso implementar para dar soporte aos operadores Grafcet de evento e temporización; no apartado (§6.4) descríbese o formato utilizado para representar os modelos Grafcet durante a súa execución na máquina virtual; no apartado (§6.5) explícase en detalle a implementación das fases do compilador; e, finalmente, no apartado (§6.6) resúmense algunhas conclusións.

6.1. O proceso de compilación

O proceso de compilación dun modelo Grafcet resúmese na Figura 6.1. O compilador recibe como entradas o modelo Grafcet a compilar —arquivo *model.sfc*— e as opcións de

configuración do proceso de compilación —arquivo *options.cfg*—. A partir destas entradas e coa asistencia de dúas aplicacións externas —*preprocess* e *sidoni*—, xéranse como resultado da compilación dous arquivos —*model.h* e *model.cpp*— que conteñen o código fonte en C++ dunha DLL. A compilación do código desta DLL realízase mediante unha terceira aplicación externa —*compile*— que xera o código binario que poderá ser cargado posteriormente na máquina virtual, e que contén a información e funcións precisas para a execución do modelo. Ademais o compilador tamén proporciona unha listaxe dos erros atopados durante a compilación —arquivo *model.err*—, no caso de habelos.

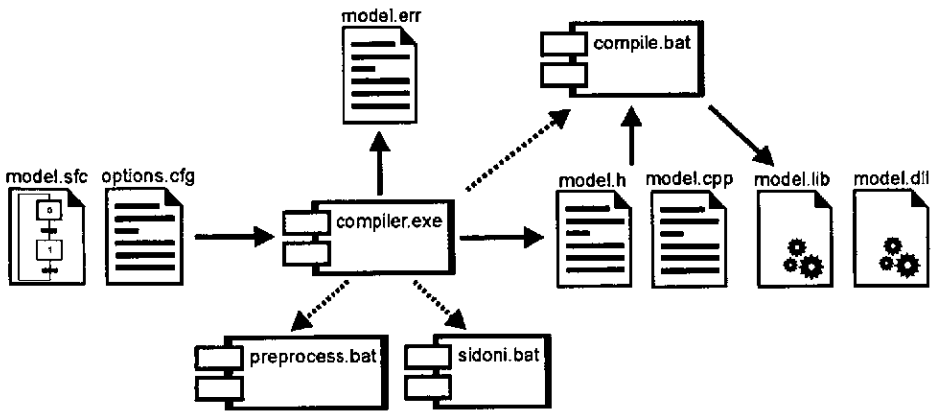


Figura 6.1. Proceso de compilación dun modelo Grafcet.

No resto desta sección explícanse certos aspectos relacionados cos arquivos de entrada e saída do compilador, coas aplicacións externas utilizadas e co proceso realizado durante a compilación.

6.1.1. Os arquivos de entrada

O compilador recibe dous arquivos de entrada:

1. *O modelo Grafcet*, creado mediante a libraría que implementa o metamodelo explicada en (§5.2). O código de accións e receptividades suponse especificado en C++ coas extensións que se detallan en (§6.3.1).
2. *As opcións de configuración*. O compilador recibe esta información mediante unha instancia da clase *SFCCompilationInfo*, cuxa declaración (simplificada) é a seguinte:

```
282. struct SFCCompilationInfo
283. {
284.   string project_name;      // proxecto Grafcet
285.   string project_path;
286.   string sfcpp_path;        // compilador Grafcet
287.   string compiler_name;     // compilador C++
288.   string iomap_file;        // declaracións E/S
289.   deque<string> user_libs; // librarías externas
290.   bool simultaneous;        // ¿eventos simultáneos?
291. }
```

A información almacenada nos atributos desta clase é a seguinte:

1. Nome e localización do proxecto —atributos *project_name* (liña 284) e *project_path* (liña 285)—.
2. Localización do compilador Grafcet —atributo *sfcpp_path* (liña 286)—.
3. Nome do compilador C++ externo —atributo *compiler_name* (liña 287)—.

4. Nome do arquivo que contén as declaracións das variábeis de proceso utilizadas —atributo *iomap_file* (liña 288)—.
5. Ubicación das librarías externas utilizadas polo usuario —atributo *user_libs* (liña 289)—.
6. Indicador booleano da posíbel ocorrencia de eventos simultáneos utilizado na simplificación de expresións con eventos (§6.5.1.5.3) —atributo *simultaneous* (liña 290)—.

Basicamente estas opcións son utilizadas para indicar a localización dos arquivos temporais, librarías e aplicacións externas utilizados durante a compilación. O compilador presupón a existencia dunha estructura de directorios como a da Figura 6.2. O executábel do compilador Grafcet está almacenado no directorio principal —atributo *sfcpp_path*—. Dentro deste directorio hai catro subdirectorios que conteñen o seguinte:

1. *include*, os arquivos de cabeceira (*.h*) das librarías utilizadas polo compilador Grafcet.
2. *lib*, as librarías utilizadas polo compilador.
3. *systemIO*, a localización por defecto dos arquivos coas declaracións das variábeis de proceso.
4. *compilers*, a información precisa para acceder ás aplicacións externas utilizadas polo compilador. Hai un subdirectorio para a aplicación Sidoni (§1.3.3) que sirve para a simplificación de expresións booleanas con eventos, e un subdirectorio por cada compilador C++ soportado polo compilador Grafcet. O compilador C++ é utilizado para o preprocesado do código de accións e receptividades e para a compilación do código fonte da DLL xerado polo compilador Grafcet.

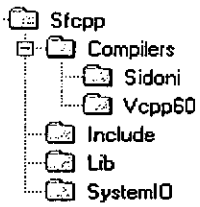


Figura 6.2. Estructura de directorios utilizada polo compilador Grafcet.

Ademais o compilador utiliza un directorio adicional para o almacenamento dos arquivos temporais e dos arquivos resultado do proceso de compilación. Este directorio crease dentro do directorio do proxecto — atributo *project_path*— co nome *compiled*.

6.1.2. O resultado da compilación

A compilación dun modelo Grafcet produce como resultado dous arquivos (Figura 6.1) que conteñen o código C++ necesario para crear, mediante un compilador externo, unha DLL que permita cargar na máquina virtual a información precisa para a execución do modelo. En (§6.5.1.9) pode verse a estructura do código xerado para unha DLL xenérica, que consta, basicamente, de tres partes:

1. O código que inicia a información do modelo durante a carga da DLL en memoria. A estructura desta información descríbese en (§6.4) e consiste nunha clase que facilita o acceso tanto á información estructural como ás funcións da DLL nas que se inclúe o código do modelo.
2. As funcións C++ que conteñen o código do modelo: accións, condicións de acción, temporizadores e condicións de transición.

3. A implementación da interface que permite a carga e descarga de módulos executábeis na máquina virtual (§7.1.1.5). A información do modelo cárgase dinamicamente na máquina virtual utilizando esta interface.

6.1.3. As aplicacións externas

Durante o proceso de compilación é preciso utilizar varias aplicacións externas que proporcionen as funcións seguintes:

1. Simplificación de expresións lóxicas con eventos (§6.5.1.5.3).
2. Preprocesamento do código C++ dacordo ás directivas de preprocesador (*#define*, *#ifdef*, *#include*, etc.) presentes no código.
3. Compilación do código fonte xerado polo compilador Grafcet e enlazado da DLL resultado coas librarías que acompañan ao compilador máis as proporcionadas polo usuario.

Durante o deseño do compilador Grafcet quíxose manter a independencia entre a súa implementación e as aplicacións externas, de xeito que se proporcionase unha técnica sinxela que permitira utilizar diferentes simplificadores de expresións lóxicas, preprocesadores, compiladores ou enlazadores sen que iso requirise modificar o código do compilador. Ademais esta característica facilitaría, en futuras versións do compilador, dar soporte a outras linguaxes diferentes ao C++. A técnica utilizada consiste en acceder dende o compilador Grafcet ás aplicacións externas a través de clases "wrapper" que executan arquivos de comandos do sistema operativo (arquivos *.bat* en Windows). Estas clases son responsábeis tamén de realizar as operacións de conversión de formatos e a creación e manexo de arquivos temporais cando sexa necesario.

Por exemplo, no caso da compilación e enlazado da DLL resultado, definiuse unha clase *SFCCPPCompiler* —derivada de *SFCCompilerPhase* (§6.2)—, que ten a interface pública seguinte:

```
292. class SFCCPPCompiler : public SFCCompilerPhase
293. {
294. public:
295.    explicit SFCCPPCompiler(Phase* phase);
296.    bool operator()();
297. };
```

Como a clase define o método *operator()*[47], as súas instancias poden utilizarse no código C++ do mesmo xeito que as funcións. O código para chamar ao compilador externo dende o compilador Grafcet sería o seguinte:

```
298. // definición e invocación do "wrapper" do compilador C++
299. SFCCPPCompiler compiler(<any_parent_phase>);
300. compiler();
```

A implementación do método *operator()* executa o arquivo de comandos denominado *compile.bat* localizado no subdirectorio do directorio *compilers* (Figura 6.2) indicado nas opcións de compilación — atributo *compiler_name*—. No caso do compilador Visual C++ este arquivo contén o seguinte:

```
301. nmake /C /S /F makedll.mak CFG=%1 DLLNAME=%2 SOURCEPATH=%3 SFCPPPATH=%4 USER_LIBS=%5
```

[47] En C++ as instancias de clases que declaran este operador denomínanse obxectos función ("function objects").

É dicir que a aplicación *nmake* executa as instruccións do arquivo *makedll.mak*[48] cos argumentos pasados pola clase *SFCCPPCompiler* na chamada a *compile.bat* (indicados como %1, %2, etc.). Estes argumentos proporcionan información sobre a versión da DLL a crear, o nome e localización dos arquivos fonte a compilar e a localización do compilador Grafcet e das librarías externas proporcionadas polo usuario.

Esta técnica permite modificar a compilación da DLL editando os arquivos *compile.bat* e *makedll.mak*, sen afectar á implementación do compilador Grafcet. Ademais pode utilizarse un compilador C++ diferente seguindo os pasos seguintes:

1. Crear un subdirectorio no directorio *compilers* (Figura 6.2) de nome igual ao do compilador C++.
2. Incluír no subdirectorio un arquivo denominado *compile.bat* cos comandos que realizan a compilación utilizando o novo compilador.
3. Pasarlle ao compilador Grafcet o nome do subdirectorio como parámetro de configuración.

A mesma técnica foi aplicada coas outras dúas aplicacións externas utilizadas. Para a simplificación de expresións lóxicas con eventos utilizouse a aplicación Sidoni (§1.3.3), accedida dende o compilador Grafcet a través da clase *SidoniWrapper*, que realiza as conversións de formatos precisas e executa o arquivo *sidoni.bat* almacenado no subdirectorio *sidoni* do directorio *compilers* (Figura 6.2). Para o preprocesamento do código C++ de accións e receptividades utilízase a clase *CPPActionPreprocessor*, que executa o arquivo *preprocess.bat* almacenado no subdirectorio do directorio *compilers* indicado nas opcións de compilación — atributo *compiler_name*—.

6.1.4. As operacións realizadas polo compilador

As principais operacións que o compilador Grafcet realiza para obter a DLL resultado a partir dun modelo Grafcet son as seguintes:

1. *Comprobación da corrección do modelo Grafcet.* O compilador Grafcet recibe como entrada un modelo creado utilizando a libraría explicada en (§5.2). Esta libraría garante que os modelos con ela construídos cumpran a regra sintáctica da alternancia etapa-transición e que non existan duplicados nos identificadores de grafcets parciais, etapas (e macroetapas), transicións e accións. O compilador comproba a coherencia da xerarquía de forzado (§3.2.2.4) e a corrección da estructura das macroexpansións (§3.2.2.2).
2. *Representación da información estructural do modelo* no formato utilizado para a súa execución (§6.4). Esta operación implica obter unha representación do modelo sen macroetapas, substituíndo estas polas súas macroexpansións, almacenar a información de grafcets parciais, etapas, transicións, accións, receptividades, etc. e substituír os identificadores alfanuméricos proporcionados polo usuario por identificadores numéricos únicos que serán os utilizados internamente durante a execución do modelo.
3. *Transformación do código de accións e receptividades.* Como se explica en (§6.3), o código de accións e receptividades non é directamente compilábel nun compilador C++ estándar, pois contén operadores de evento e temporización que non son parte da linguaxe C++. É preciso polo tanto, substituír os eventos e temporizadores polo código que implementa o seu funcionamento, o cal forma parte dunha das librarías do compilador. O mesmo pode dicirse do acceso aos valores das variábeis do modelo e de proceso que son

[48] Os contidos do arquivo *makedll.mak* para o compilador Visual C++ poden verse no Anexo C.

almacenadas nunha base de datos na máquina virtual (§7.3.1) durante a execución do modelo. Os seus nomes teñen que ser substituídos polas chamadas ás funcións da libraría que proporcionan acceso á base de datos en tempo de execución. As transformacións que se realizan no código son explicadas en detalle en (§6.3.3).

4. *Xeración do código fonte da DLL (en C++).*
5. *Compilación e enlazado da DLL resultado* coas librarías do compilador e máis as proporcionadas polo usuario, utilizando un compilador C++ externo.

6.2. A estructura do compilador Grafcet

A arquitectura do compilador Grafcet está formada por un conxunto de fases organizadas xerarquicamente formando un grafo dirixido acíclico (Figura 6.6) máis a información xerada durante o proceso de compilación á que todas as fases teñen acceso. Cada fase é responsábel de realizar unha operación concreta, accedendo (e modificando si é preciso) esa información. As fases do compilador execútanse secuencialmente, de xeito que cada fase utiliza a información xerada polas fases previas e prepara a que utilizarán as fases posteriores. Cada fase pode ser simple ou composta (formada por múltiples subfases), de modo que unha mesma subfase pode estar contida en múltiples fases compostas e, en consecuencia, ser executada varias veces en fases diferentes. A Figura 6.3 mostra o diagrama de clases que modela os aspectos básicos que permiten construír esta estructura.

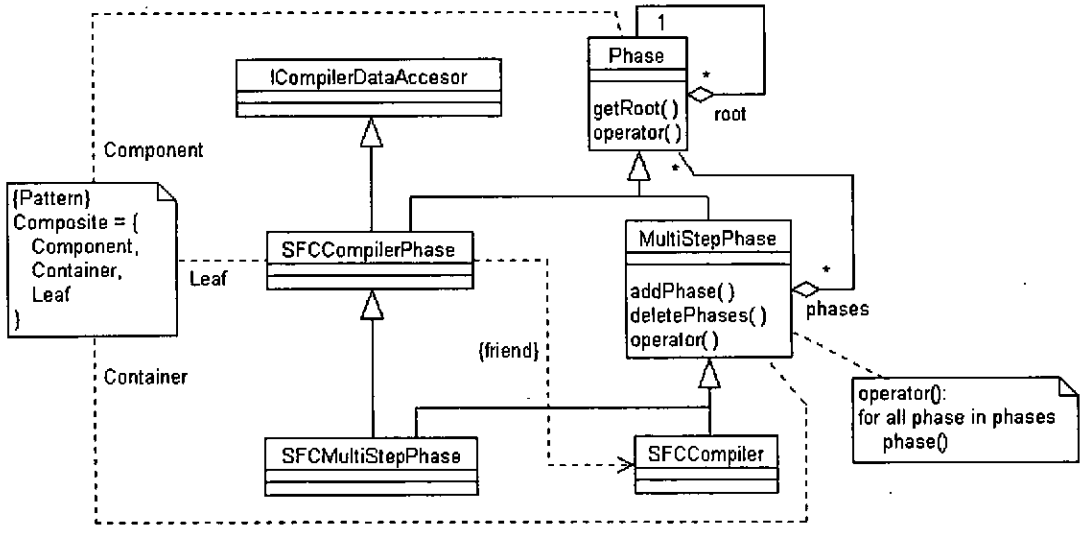


Figura 6.3. Diagrama de clases da estructura de fases do compilador Grafcet.

A relación de agregación entre fases modelouse aplicando o patrón de deseño *Composite* [67]. A clase *Phase* proporciona a interface compartida por todas as fases, que poden ser simples —clase *SFCCompilerPhase*— ou compostas —clase *MultiStepPhase*—. A clase *SFCCompiler*, derivada de *MultiStepPhase*, é a raíz do grafo dirixido acíclico que forman as fases do compilador. Esta clase proporciona un punto de acceso común ás operacións que o compilador realiza (aplicación do patrón de deseño *Facade* [67]) e almacena a información interna accedida e modificada polas fases durante a compilación. Nos apartados seguintes descríbese a información almacenada polo compilador, a súa estructura de fases e as técnicas utilizadas para a creación e execución das fases así como o acceso destas á información do compilador.

6.2.1. A información interna do compilador

O código seguinte mostra a parte da declaración da clase *SFCCompiler* na que se define a información utilizada internamente durante a compilación dun modelo Grafcet:

```
302. class SFCCompiler : public MultiStepPhase
303. {
304.  friend class SFCCompilerPhase;
305. private:
306.    // entradas ao compilador
307.    const SFCGlobal* sfc_model;      // modelo Grafcet
308.    SystemDataDeclSeq systemIO;      // declaracións das E/S do proceso
309.    SFCCompilationInfo* comp_info;   // opcións de compilación
310.    // estructura do modelo Grafcet
311.    RTSituation initial;             // situación inicial
312.    deque<RTPGInfo*> pgs;            // grafcets parciais
313.    deque<RTNodeInfo*> nodes;        // nodos
314.    deque<RTActionInfo*> actions;    // accións
315.    deque<RTFOInfo*> forders;        // ordes de forzado
316.    FOHierarchyLevels fo_levels;     // niveis da xerarquía de forzado
317.    deque<RTTimerInfo*> timers;      // temporizadores
318.    // código do modelo Grafcet
319.    SFCActionSeq actions_code;       // accións
320.    deque<RTReceptivityInfo*> recs; // receptividades
321.    // información interna do compilador
322.    deque<string> ids;               // identificadores internos
323.    set<unsigned long> used_ids;
324.    SFCCompilerPaths paths;          // arquivos e localizacións auxiliares
325.    SFCCompilerErrors* errors;       // erros de compilación
326.    SFCCompilationTempData temp_data;
327. };
```

Esta información está formada por:

1. *A información de entrada ao compilador*:
   a. O modelo Grafcet a compilar —atributo *sfc_model* (liña 307)—, no formato creado coa libraría explicada en (§5.2).
   b. As declaracións de variábeis do proceso[49] —atributo *systemIO* (liña 308)— utilizadas no código de accións e receptividades.
   c. As opcións de compilación (§6.1.1) —atributo *comp_info* (liña 309)—.
2. *A información estructural do modelo* (liñas 311-317), utilizando o formato para a representación do modelo en tempo de execución (§6.4).
3. *O código C++ de accións e receptividades* (liñas 319-320).
4. *Outras informacións auxiliares*:
   a. A información para a asignación e conversión entre os identificadores alfanuméricos únicos asignados polo usuario ás diferentes compoñentes do modelo e os numéricos utilizados en tempo de execución —atributos *ids* (liña 322) e *used_ids* (liña 323)—.
   b. A localización dos arquivos temporais e aplicacións auxiliares utilizadas durante a compilación —atributo *paths* (liña 324)—.
   c. A listaxe de erros producidos durante a compilación —atributo *errors* (liña 325)—.
   d. A información temporal utilizada para configurar a execución das fases do compilador e almacenar os resultados intermedios (§6.2.3) —atributo *temp_data* (liña 326)—.

[49] A declaración das variábeis internas están incluídas no modelo Grafcet.

6.2.2. Acceso á información interna do compilador

Para acceder á información interna almacenada polo compilador, cada fase mantén unha referencia —atributo *root* da clase *Phase* (Figura 6.3)— á raíz do grafo de fases do compilador (que é unha instancia da clase *SFCCompiler*) e implementa a interface *ICompilerDataAccesor* declarada como[50]:

```
328. struct ICompilerDataAccesor
329. {
330.    // entrada ao compilador
331.    virtual const SFCGlobal& model() const = 0;
332.    virtual const SystemDataDeclSeq& getSystemIO() const = 0;
333.    virtual const SFCCompilationInfo& getOptions() const = 0;
334.    // estructura do modelo Grafcet
335.    virtual RTSituation& initialSituation() = 0;
336.    virtual deque<RTPGInfo*>& pgs() = 0;
337.    virtual deque<RTNodeInfo*>& nodes() = 0;
338.    virtual deque<RTActionInfo*>& actions() = 0;
339.    virtual deque<RTFOInfo*>& forders() = 0;
340.    virtual FOHierarchyLevels& folevels() = 0;
341.    virtual deque<RTTimerInfo*>& timers() = 0;
342.    // código do modelo Grafcet
343.    virtual SFCActionSeq& actionscode() = 0;
344.    virtual deque<RTReceptivityInfo*>& receptivities() = 0;
345.    // métodos auxiliares
346.    virtual unsigned long getId(const string& key) = 0;
347.    virtual void registerId(unsigned long id) = 0;
348.    virtual bool isIdUsed(unsigned long id) const = 0;
349.    virtual SFCCompilerPaths& getPaths() = 0;
350.    virtual void signalError(SFCCompilerErrMsg* err) = 0;
351.    virtual SFCCompilationTempData& getTempData() = 0;
352. };
```

Esta interface impleméntase na clase *SFCCompilerPhase* da que derivan todas as fases simples do compilador. Esta clase é unha 'amiga' (Figura 6.3) da clase *SFCCompiler*, polo que ten acceso aos seus membros privados (entre eles a información interna do compilador). A implementación por defecto dos métodos da interface *ICompilerDataAccesor* accede directamente ao atributo do compilador no que está a información requirida. O seguinte código mostra, a modo de exemplo, como sería a implementación dun destes métodos:

```
353. SFCCompilationTempData& SFCCompilerPhase::getTempData()
354. {
355.   SFCCompiler* pRoot = dynamic_cast<SFCCompiler*>( getRoot() );
356.   assert( pRoot != NULL );
357.   return pRoot->temp_data;
358. }
```

Este método de acceso permite que as fases redefinan estes métodos para realizar as operacións requiridas de iniciación, comprobación ou formatado da información do compilador antes de ser utilizada pola fase. O seguinte código mostra un exemplo de redefinición do método anterior para unha fase hipotética:

```
359. SFCCompilationTempData& AnySimplePhase::getTempData()
360. {
361.   SFCCompilationTempData& data = SFCCompilerPhase::getTempData();
362.   // iniciar, comprobar ou modificar aquí a información antes de utilizala
363.   return data;
364. }
```

[50] A interface mostrada é unha versión reducida. Na interface completa decláranse dúas versións de cada método, unha a mostrada, e outra unha versión constante (p.e. *virtual const RTSituation& initialSituation() const = 0;*).

6.2.3. As fases do compilador

A Figura 6.4 e a Figura 6.5 mostran os diagramas de clases das fases definidas no compilador Grafcet. As fases simples son derivadas da clase *SFCCompilerPhase* e as compostas de *SFCMultiStepPhase*. Cada unha das clases que modela unha fase, xa sexa simple ou composta, redefine o operador *operator()* no que se implementan as operacións propias de cada fase.

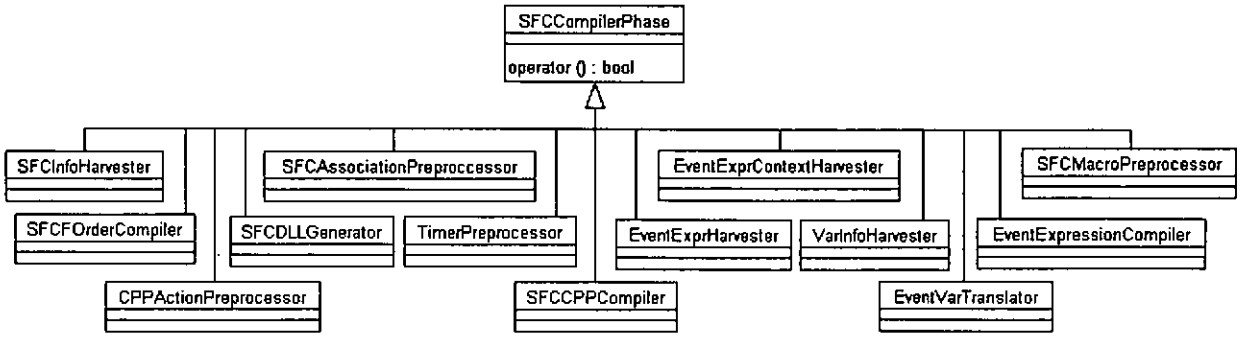


Figura 6.4. Diagrama de clases das fases simples do compilador Grafcet.

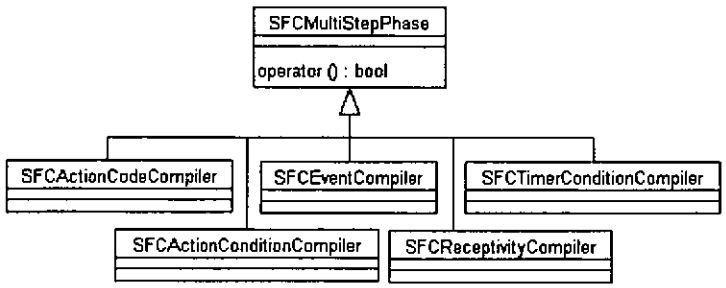


Figura 6.5. Diagrama de clases das fases compostas do compilador Grafcet.

Durante a iniciación do compilador constrúese a estructura de fases da Figura 6.6, utilizando a técnica explicada en (§6.2.4). Esta estructura é un grafo dirixido acíclico no que as fases compostas *SFCActionCodeCompiler*, *SFCActionConditionCompiler*, *SFCReceptivityCompiler* e *SFCTimerConditionCompiler* comparten as mesmas subfases[51].

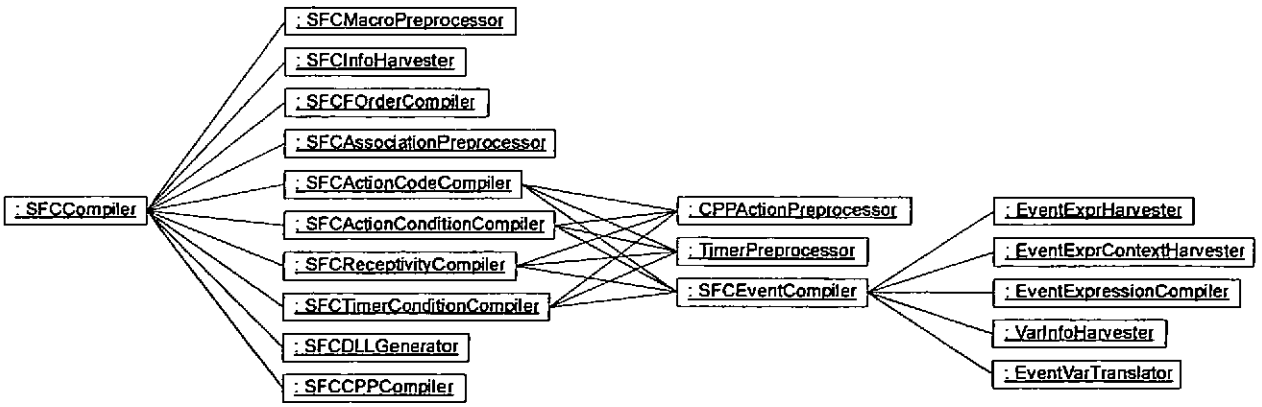


Figura 6.6. Estructura de fases do compilador Grafcet.

[51] A orde de execución das fases móstrase na figura de esquerda a dereita e de arriba a baixo

Nas catro fases indicadas é nas que se compila respectivamente o código C++ das accións, condicións das asociacións, condicións de transición e condicións dos temporizadores. As operacións realizadas en cada caso son moi semellantes, polo que durante o deseño decidiuse compartir a estructura de fases e configurar as opcións específicas de cada caso mediante a información almacenada no compilador, nunha instancia da clase *SFCCompilationTempData*. A interface pública desta clase é a seguinte:

```
365. struct SFCCompilationTempData
366. {
367.  // "buffers" temporais para almacenar as transformacións no código
368.  stringstream original_code;
369.  stringstream preprocessed_code;
370.  stringstream modified_code;
371.  stringstream final_code;
372.  // opcións temporais para configurar as fases do compilador
373.  bool in_condition;
374.  bool timers_allowed;
375.  // información de expresións e variábeis
376.  ExpressionSeq event_expressions;
377.  VarInfoSeq var_info;
378.  // métodos públicos
379.  void initialize(const string& code, bool cond, bool timers);
380.  void clear();
381. };
```

Os atributos que se declaran na clase *SFCCompilationTempData* son:

1. *original_code* (liña 368), *preprocessed_code* (liña 369), *modified_code* (liña 370) e *final_code* (liña 371), "buffers" utilizados para almacenar os cambios que as subfases realizan no código orixinal para transformalo nun código executábel na máquina virtual.
2. *in_condition* (liña 373), valor booleano que indica se o código que se está a compilar é o dunha condición ou o dunha acción. Algunhas das transformacións a realizar (§6.3.3) dependen deste valor, por exemplo, non está permitido modificar variábeis nas condicións.
3. *timers_allowed* (liña 374), valor booleano que indica se está permitida a utilización de temporizadores no código que se está a compilar. Na versión actual do compilador unicamente se permite a utilización de temporizadores nas condicións de transición.
4. *event_expressions* (liña 376) e *var_info* (liña 377), información temporal sobre expresións e variábeis utilizada para realizar transformacións no código segundo se explica en (§6.3.3).

Ademais decláranse os métodos *initialize* (liña 379) e *clear* (liña 380) para establecer os valores iniciais destes atributos. O seguinte pseudocódigo mostra a implementación xenérica do operador *operator()* nas fases indicadas anteriormente:

```
382. AnyCompoundPhaseClass::operator()()
383. {
384.  for every code_block code
385.   /* incluír aquí o código de preprocesamento específico da fase */
386.   // iniciar información de configuración
387.   getTempData().initialize(code, <in_condition>, <timers_allowed>);
388.   // executar as subfases
389.   SFCMultiStepPhase::operator()();
390.   /* incluír aquí o código de postprocesamento específico da fase */
391.   code = getTempData().final_code;  // almacenar código transformado
392.   getTempData().clear();            // reiniciar información auxiliar
393.  end for
394. }
```

O código fonte a compilar está almacenado na variábel *code*. Cada fase composta concreta iniciará a información de configuración (liña 387) con este código e os valores correspondentes dos argumentos *in_condition* e *timers_allowed* indicados na Táboa 6-I. Unha vez executadas as subfases (liña 389), o código modificado é almacenado na variábel *code* (liña 391) e a información temporal reiniciada (liña 392). Isto repítese para cada bloque de código que haxa que compilar (p.e. a fase *SFCReceptivityCompiler* repite a execución das liñas 384-393, unha vez por cada condición de transición).

	Condición	Temporizadores
SFCActionCodeCompiler	Non	Non
SFCActionConditionCompiler	Si	Non
SFCReceptivityCompiler	Si	Si
SFCTimerConditionCompiler	Si	Non

Táboa 6-I. Valores dos atributos *in_condition* e *timers_allowed* nas fases compostas do compilador Grafcet.

6.2.4. Iniciación e execución das fases do compilador

Como xa foi comentado anteriormente, a estructura de fases do compilador organízase nunha xerarquía en forma de grafo dirixido acíclico (Figura 6.6). Cada fase pode ser simple ou estar formada por múltiples subfases, que poden ser compartidas entre varias fases compostas. Estas son responsábeis de configurar e iniciar a execución das subfases que conteñen. A orde de execución é secuencial e a estructura de fases é estática (non se modifica durante a execución). A raíz da estructura é unha instancia da clase *SFCCompiler*, que é a responsábel de crear e almacenar o resto da estructura durante a iniciación do compilador. O código seguinte mostra a declaración da clase abstracta *Phase* (Figura 6.3), da que derivan todas as fases do compilador:

```
395. class Phase
396. {
397. private:
398.   Phase* root;
399. public:
400.   virtual Phase* getRoot() const { return root; }
401.   virtual bool operator()() = 0;
402. };
```

A clase contén un único atributo —*root* (liña 398)— no que se almacena a referencia á raíz da estructura de fases. O valor deste atributo é iniciado no constructor e accedido mediante o método *getRoot* (liña 400). A execución da fase implementouse mediante o operador C++ *operator()*, de xeito que se utilice da mesma maneira que a chamada a unha función. Este operador é redefinido nas clases derivadas para implementar as operacións realizadas en cada fase concreta. As fases compostas son derivadas da clase abstracta *MultiStepPhase* (Figura 6.3), que é declarada do xeito seguinte:

```
403. class MultiStepPhase : public virtual Phase
404. {
405. private:
406.   list<Phase*> phases;
407. public:
408.   void add_phase(Phase* phase, bool back = true);
409.   void delete_phases();
410.   virtual bool operator()();
411. };
```

Esta clase, derivada da clase *Phase*, declara un atributo —*phases* (liña 406)— no que se almacena a secuencia de subfases, e dous métodos —*add_phase* (liña 408) e *delete_phases* (liña 409)— para engadir e eliminar subfases. Ademais esta clase inclúe unha implementación por defecto para o operador *operator()* que executa as subfases secuencialmente:

```
412. bool MultiStepPhase::operator()()
413. {
414.  bool error = false;
415.  for every subphase in phases
416.    error = subfase.operator()(); // executar a subfase
417.  end for
418.  return error;
419. }
```

A Figura 6.7 mostra de forma simplificada a secuencia de mensaxes que se intercambian entre a fase principal do compilador e as subfases durante o proceso de compilación.

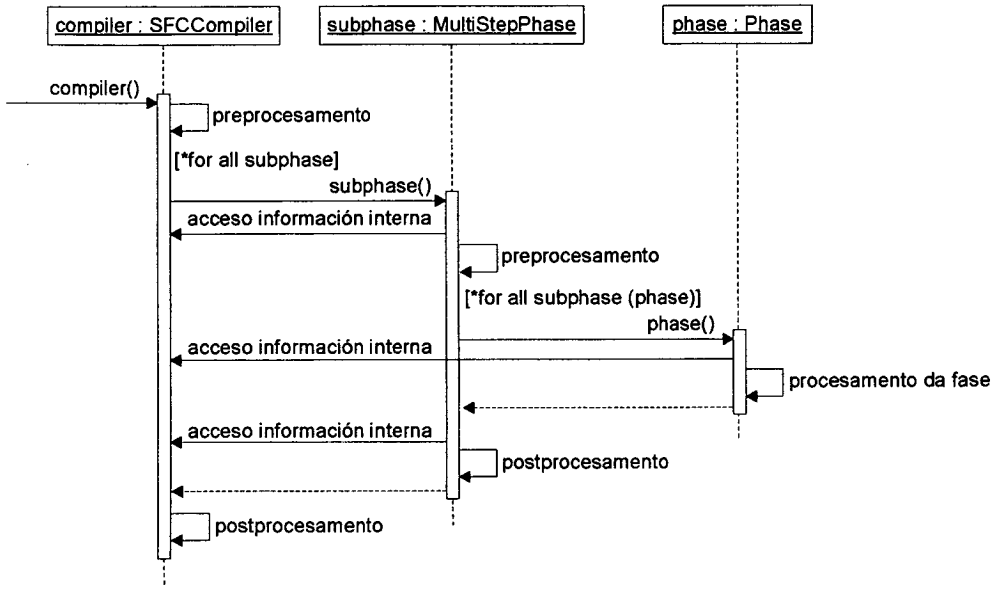


Figura 6.7. Secuencia das mensaxes intercambiadas entre a fase principal e as subfases do compilador Grafcet.

A creación da estructura de fases faise durante a iniciación do compilador, no constructor das fases compostas (a maior parte da estructura iníciase no constructor da clase *SFCCompiler*). O seguinte código mostra como se crea a estructura de fases no constructor dunha fase composta hipotética:

```
420. AnyCompoundPhaseClass::AnyCompoundPhaseClass()
421. {
422.  // engadir unha subfase simple
423.  Phase* phase = new AnySimplePhaseClass(this);
424.  add_phase(phase);
425.  // engadir unha subfase composta
426.  MultiStepPhase* multiphase = new AnyCompoundPhaseClass(this);
427.  add_phase(multiphase);
428.  // engadir unha subfase simple á subfase composta
429.  Phase* subphase = new AnySimplePhaseClass(this);
430.  multiphase->add_phase(subphase);
431. }
```

6.3. Consideracións sobre a utilización de C++ nos modelos Grafcet

O Grafcet foi concibido inicialmente [1] como un formalismo gráfico para a especificación de controladores lóxicos secuenciais. Os modelos Grafcet describen os valores que toma un conxunto de saídas en función dos valores dun conxunto de entradas, variábeis internas e da situación do modelo, determinada polo conxunto de etapas activas. A situación do modelo evoluciona dacordo a un conxunto de condicións lóxicas asociadas as transicións, nas que se relacionan os valores das entradas, as variábeis internas e os estados das etapas. Todas as variábeis utilizadas no modelo son booleanas.

Nesta descrición básica do modelo, cada acción asociada a unha etapa unicamente modifica o valor dunha saída booleana dacordo a unha semántica ben definida que depende do seu tipo (§3.3.3). Os únicos elementos do modelo que aceptan unha representación textual son as condición booleanas asociadas a transicións e accións (nas accións condicionais). O estándar non propón unha sintaxe determinada para a especificación das condicións lóxicas, indicando unicamente que poderán utilizarse os operadores booleanos: AND, OR e NOT, os operadores de evento (↑, ↓) e o de retardo (t1\Xi\t2).

A adopción dunha versión do Grafcet como linguaxe de programación de PLCs (§3.6) requiriu dúas consideracións en relación a este modelo básico:

1. A utilización de variábeis non booleanas para dar soporte aos diferentes tipos de datos e cálculos manexados nos PLCs.
2. A posibilidade de especificar código nas accións alén das accións booleanas básicas do modelo. O SFC permite definir bloques de código cuxa execución é controlada polas accións asociadas ás etapas dacordo á semántica definida por un bloque de control de acción (§3.6.1.3). O contido das accións e condicións lóxicas pode especificarse utilizando diferentes linguaxes tanto textuais como gráficas: IL, ST, LD e FBD.

No metamodelo proposto en (§5.1) adóptanse, entre outras, estas dúas extensións ao modelo Grafcet orixinal, e definense os conceptos relacionados coa representación dos bloques de código e os tipos de accións—os aspectos relacionados coa súa execución explícanse en (§8.6)—. O metamodelo deixa aberta a posibilidade de utilizar calquera linguaxe para a especificación dos contidos de condicións lóxicas e bloques de acción. O soporte á utilización de C++ implementado no compilador proporciona as seguintes avantaxes:

1. Dispone de todos os tipos de datos e operadores da linguaxe.
2. Poden utilizarse as librarías externas existentes para incluír as funcionalidades que se precisen.
3. As características de orientación a obxecto da linguaxe permiten estructurar e reutilizar o código.
4. Resulta simple incluír funcións avanzadas como, por exemplo, a detección de fallas baseada na execución dun modelo "software" do proceso controlado.

Sen embargo hai un inconveniente importante na utilización do C++: os operadores Grafcet de evento (↑, ↓) e retardo (t1\Xi\t2) non son parte da linguaxe. En consecuencia é preciso implementar soporte para estes operadores, o que implica:

1. Estender a sintaxe das expresións C++ para engadir os novos operadores.
2. Proporcionar a infraestructura (en forma de libraría de funcións) que implemente a semántica dos operadores durante a execución dos modelos.

3. Transformar as expresións estendidas en expresións C++ estándar durante a compilación do modelo (cada operador é substituído polo código que accede ás funcións que implementan a súa semántica).

Unha consideración semellante afecta ao uso de variábeis do modelo (entradas, saídas, variábeis internas, nomes de etapa, etc.) no código C++. Aínda que sintacticamente o seu uso é igual ao de calquera outra variábel C++, o acceso ao seu valor durante a execución do modelo é diferente. Os valores das variábeis C++ (declaradas no código) almacénanse na memoria do programa durante a execución do modelo, sen embargo, o valor das variábeis do modelo (declaradas como parte do modelo) é almacenado durante a execución nunha base de datos na máquina virtual (§7.3.1). A obtención do seu valor faise mediante funcións que implementan o acceso á base de datos en tempo de execución. Polo tanto, durante a compilación é preciso substituír os nomes das variábeis do modelo polo código que utilice estas funcións.

No resto desta sección explícanse as modificacións realizadas na sintaxe e gramática do C++ para incluír os operadores de evento e temporización, o soporte proporcionado pola máquina virtual para implementar a súa semántica e as técnicas utilizadas para substituír estes operadores polo código que implementa a súa funcionalidade.

6.3.1. Extensión da sintaxe C++ para incluír os operadores Grafcet

Neste apartado explícanse en detalle as modificacións realizadas dende o punto de vista sintáctico para incluír os operadores Grafcet de evento e temporización na linguaxe C++. Estas modificacións consisten en:

1. Definición da sintaxe dos operadores.
2. Modificación da gramática das expresións C++ para incluír os novos operadores.

6.3.1.1. Sintaxe dos operadores Grafcet

Os *operadores de evento* (↑ e ↓) defínironse como operadores booleanos unarios en C++ (como o *NOT* lóxico, por exemplo). Estes operadores poden utilizarse directamente con variábeis booleanas (↑*bool_var*) ou con expresións booleanas entre parénteses[52] (↑(*expr*)). As expresións con eventos poden aniñarse, tal e como mostran os seguintes exemplos:

↑a || ↓d && !(↓(3*b > 12))
↑(a && ↓(b == 2) || ↑(↓d))

O *operador de temporización* (T#) representa a semántica do operador de retardo definido en [83]. A condición de temporización foi estendida para permitir a utilización de expresións C++ complexas. A sintaxe completa do operador é a seguinte:

T#(<*t1_expr*>; <*cond_expr*>; <*t2_expr*>)

Os argumentos da temporización están separados polo delimitador (;) e son todos opcionais, do mesmo xeito que na instrucción *for* do C++. O significado destes argumentos é o seguinte:

1. *t1_expr*, expresión numérica constante que indica o tempo de retardo que a condición do temporizador ten que ser certa para que este se active. O valor por defecto é cero.

[52] Como se explica en (§6.5.1.5.3), a restricción de que a expresión vaia entre parénteses simplifica a implementación da fase *EventExprHarvester*.

2. *cond_expr*, expresión booleana que activa o temporizador. O valor por defecto é *true*.
3. *t2_expr*, expresión numérica constante que indica o tempo que o temporizador permanece activo despois de que a condición deixe de ser certa. O valor por defecto é cero.

Os operadores de temporización non poden aniñarse, é dicir, non poden utilizarse temporizadores nas condicións doutros temporizadores. Os seguintes exemplos mostran algunhas expresións con temporizadores:

T#(; a&&b>20||↓d; 350)
T#(30; !a;)
T#(150; ; 300)

6.3.1.2. Modificación da gramática do C++

A gramática C++ utilizada como base é a ANSI [88]. Os cambios realizados consistiron en engadir "tokens" léxicos para representar os novos operadores (↑, ↓ e T#) e en modificar a regra que afecta á sintaxe das expresións unarias para incluír os dous novos tipos de expresións[53]:

```
432. Expressions [gram.expr]
433.
434. unary-expression:
435.    postfix-expression
436.    ++ cast-expression
437.    -- cast-expression
438.    unary-operator cast-expression
439.    sizeof unary-expression
440.    sizeof ( type-id )
441.    new-expression
442.    delete-expression
443.    sfcpp-event-expression                                    [+]
444.    sfcpp-timer-expression                                    [+]
445.
446. sfcpp-event-expression:                                      [+]
447.    ↑ identifier                                              [+]
448.    ↓ identifier                                              [+]
449.    ↑ ( sfcpp-expression )                                    [+]
450.    ↓ ( sfcpp-expression )                                    [+]
451.
452. sfcpp-timer-expression:                                      [+]
453.    T# ( digit-sequence_opt ; sfcpp-expression_opt ; digit-sequence_opt )   [+]
```

As regras *sfcpp-event-expression* (liña 446) e *sfcpp-timer-expression* (liña 452) definen a sintaxe das novas expresións. Nótese que na definición destas regras utilízase unha versión simplificada das expresións C++ —regra *sfcpp-expression* (liña 454)—. Estas expresións teñen as seguintes restricción con respecto ás expresións C++:

- Non se permiten as conversións de tipo explícitas nin o uso dos operadores de conversión *dynamic_cast*, *static_cast*, *reinterpret_cast* e *const_cast*.
- Non se permite o uso dos operadores *sizeof*, *typeid*, *new* e *delete*.

[53] Por conveniencia reprodúcense unicamente as regras da gramática C++ afectadas polas modificacións. Estas están indicadas en negriña e cun símbolo á dereita da páxina indicando o tipo de modificación: [+] parte engadida, [-] parte eliminada e [*] parte modificada. En caso de requerirse algún comentario adicional indícase tamén a referencia numérica das notas incluidas ao final da gramática.

- Non se permite o uso da palabra reservada *throw*, polo que dende as expresións simplificadas non poden lanzarse excepcións.
- Non se permite o uso das palabras reservadas *typename* e *template*, polo que nas expresións simplificadas non pode usarse a cualificación explícita dos tipos de datos nin dos membros de "templates".
- Non se permite o uso da palabra reservada *operator*, polo que nas expresións simplificadas non poden referenciarse explicitamente os operadores declarados polo usuario.
- Inclúense as expresións con eventos e temporizadores do Grafcet.

A razón de utilizar estas expresións simplificadas é que o número de regras sintácticas necesarias para representar as expresións redúcese de maneira significativa, o que simplifica a implementación da fase *EventExpressionCompiler* (§6.5.1.5.3). Ademais na práctica estas restriccións non supoñen un problema importante, pois o uso dalgunha destas características en expresións con eventos e temporizadores non tería sentido (p.e. o uso de *delete* ou *throw*) e, en último caso, sempre podería buscarse unha representación alternativa da expresión que utilizase estas características de forma indirecta a través de variábeis ou funcións auxiliares.

En consecuencia, tomouse a decisión de primar a simplicidade do compilador fronte ao soporte das características indicadas, que poderán engadirse en futuras versións. A gramática completa das expresións simplificadas toma como base a das expresións C++, e é a seguinte[54]:

```
454. sfcpp-expression:
455.    sfcpp-assignment-expression
456.    sfcpp-expression, sfcpp-assignment-expression                    [-]
457.
458. sfcpp-expression-list:
459.    sfcpp-expression
460.    sfcpp-expression, sfcpp-expression-list                          [+]
461.
462. sfcpp-assignment-expression:
463.    sfcpp-conditional-expression
464.    sfcpp-logical-or-expression assignment-operator sfcpp-assignment-expression
465.    throw-expression                                                 [-]
466.
467. assignment-operator: one of
468.    =  *=  /=  %=  +=  -=  >>=  <<=  &=  ^=  |=
469.
470. sfcpp-conditional-expression:
471.    sfcpp-logical-or-expression
472.    sfcpp-logical-or-expression ? sfcpp-expression : sfcpp-assignment-expression
473.
474. sfcpp-logical-or-expression:
475.    sfcpp-logical-and-expression
476.    sfcpp-logical-or-expression || sfcpp-logical-and-expression
477.
478. sfcpp-logical-and-expression:
479.    sfcpp-inclusive-or-expression
480.    sfcpp-logical-and-expression && sfcpp-inclusive-or-expression
481.
482. sfcpp-inclusive-or-expression:
483.    sfcpp-exclusive-or-expression
484.    sfcpp-inclusive-or-expression | sfcpp-exclusive-or-expression
485.
486. sfcpp-exclusive-or-expression:
487.    sfcpp-and-expression
488.    sfcpp-exclusive-or-expression ^ sfcpp-and-expression
```

[54] As regras sintácticas das expresións simplificadas teñen o mesmo nome que as do C++ co prefixo *sfcpp-*.

```
sfcpp-and-expression:
   sfcpp-equality-expression
   sfcpp-and-expression & sfcpp-equality-expression

sfcpp-equality-expression:
   sfcpp-relational-expression
   sfcpp-equality-expression == sfcpp-relational-expression
   sfcpp-equality-expression != sfcpp-relational-expression

sfcpp-relational-expression:
   sfcpp-shift-expression
   sfcpp-relational-expression < sfcpp-shift-expression
   sfcpp-relational-expression > sfcpp-shift-expression
   sfcpp-relational-expression <= sfcpp-shift-expression
   sfcpp-relational-expression >= sfcpp-shift-expression

sfcpp-shift-expression:
   sfcpp-additive-expression
   sfcpp-shift-expression << sfcpp-additive-expression
   sfcpp-shift-expression >> sfcpp-additive-expression

sfcpp-additive-expression:
   sfcpp-multiplicative-expression
   sfcpp-additive-expression + sfcpp-multiplicative-expression
   sfcpp-additive-expression - sfcpp-multiplicative-expression

sfcpp-multiplicative-expression:
   sfcpp-pm-expression
   sfcpp-multiplicative-expression * sfcpp-pm-expression
   sfcpp-multiplicative-expression / sfcpp-pm-expression
   sfcpp-multiplicative-expression % sfcpp-pm-expression

sfcpp-pm-expression:
   cast-expression                                          [*:1]
   sfcpp-pm-expression .* cast-expression                   [*:1]
   sfcpp-pm-expression ->* cast-expression                  [*:1]

cast-expression:                                            [-:1]
   sfcpp-unary-expression
   ( type-id ) cast-expression                              [-:1]

sfcpp-unary-expression:
   sfcpp-postfix-expression
   ++  cast-expression                                      [*:1]
   --  cast-expression                                      [*:1]
   unary-operator cast-expression                           [*:1]
   sizeof unary-expression                                  [-]
   sizeof ( type-id )                                       [-]
   new-expression                                           [-]
   delete-expression                                        [-]
   sfcpp-event-expression                                   [+]
   sfcpp-timer-expression                                   [+]

unary-operator: one of
   *  &  +  -  !  ~

sfcpp-event-expression:                                     [+]
   ↑ identifier                                             [+]
   ↓ identifier                                             [+]
   ↑ ( sfcpp-expression )                                   [+]
   ↓ ( sfcpp-expression )                                   [+]

sfcpp-timer-expression:                                     [+]
   T# ( digit-sequence_opt ; sfcpp-expression_opt ; digit-sequence_opt )   [+]
```

```
554. sfcpp-postfix-expression:
555.    sfcpp-primary-expression
556.    postfix-expression [ sfcpp-expression ]                          [*:2]
557.    postfix-expression ( sfcpp-expression-list_opt )                 [*:2]
558.    simple-type-specifier ( expression-listopt )                     [-]
559.    postfix-expression . template_opt ::_opt sfcpp-id-expression     [*:2,3]
560.    postfix-expression -> template_opt ::_opt sfcpp-id-expression    [*:2,3]
561.    postfix-expression . pseudo-destructor-name                      [-]
562.    postfix-expression -> pseudo-destructor-name                     [-]
563.    postfix-expression ++                                            [*:2]
564.    postfix-expression --                                            [*:2]
565.    dynamic_cast < type-id > ( expression )                          [-]
566.    static_cast < type-id > ( expression )                           [-]
567.    reinterpret_cast < type-id > ( expression )                      [-]
568.    const_cast < type-id > ( expression )                            [-]
569.    typeid ( expression )                                            [-]
570.    typeid ( type-id )                                               [-]
571.
572. sfcpp-primary-expression:
573.    literal
574.    this
575.    :: identifier                                                    [-]
576.    :: operator-function-id                                          [-]
577.    :: qualified-id                                                  [-]
578.    ( sfcpp-expression )
579.    sfcpp-id-expression
580.    identifier sfcpp-id-expression_opt                               [+:4]
581.
582. sfcpp-id-expression:                                               [+:4]
583.    :: identifier sfcpp-id-expression_opt                            [+:4]
584.
585. digit-sequence: igual que en C++
586. literal: igual que en C++
587. identifier: igual que en C++
```

Notas:

1. A regra *cast_expression* é eliminada na gramática simplificada e as ocorrencias deste símbolo substituídas por *sfcpp-unary-expression*.
2. As ocorrencias do símbolo *postfix-expression* son substituídas por *sfcpp-primary-expression* na gramática simplificada.
3. As partes optativas destas regras son eliminadas na gramática simplificada.
4. A regra *sfcpp-id-expression* simplificouse para non permitir a cualificación explícita de tipos de datos, membros de "templates" nin operadores declarados polo usuario.

6.3.2. Implementación dos operadores Grafcet

Como se explicou na introducción desta sección, a implementación da semántica dos operadores de evento e temporización é implementada substituíndoos por chamadas a métodos dunha interface implementada na máquina virtual (§8.6), dacordo ao explicado en (§6.3.3). Esta interface permite acceder aos valores das variábeis e comprobar o estado de eventos e temporizadores durante a execución dos modelos Grafcet. A súa declaración é a seguinte:

```
588. struct IVMachineAccess
589. {
590.   // variábeis internas
591.   virtual bool getModelVarEvent(const string& id, bool updown) const = 0;
592.   template <class T>
593.     bool getModelVar(const string& id, T& value) const;
594.   template <class T>
595.     bool setModelVar(const string& id, const T& value);
```

```
596.   // variábeis de proceso
597.   virtual bool getSystemVarEvent(const string& id, bool updown) const = 0;
598.   template <class T>
599.    bool getSystemVar(const string& id, T& value) const;
600.   template <class T>
601.    bool setSystemVar(const string& id, const T& value);
602.   // estado de etapas e temporizadores
603.   virtual bool getStepState(unsigned long id) const = 0;
604.   virtual bool getTimerState(const string& id) const = 0;
605. };
```

Nótese que o acceso aos valores das variábeis internas (liñas 591-595) e de proceso (liñas 597-601) está representado mediante métodos parametrizados (métodos *template* en C++) cuxo parámetro é o tipo de dato da variábel accedida. Todos o métodos utilizan o identificador alfanumérico da variábel ou temporizador coa excepción do método que accede ao estado das etapas do modelo (liña 603) que utiliza o identificador numérico asignado automaticamente polo compilador. Ademais os métodos que consultan a ocorrencia de eventos (liñas 591 e 597) teñen un segundo parámetro booleano que indica o tipo de evento consultado (flanco positivo ou negativo do sinal). Nótese tamén que a ocorrencia de eventos, o estado das etapas e o estado dos temporizadores unicamente poden ser consultados (os seus valores son modificados internamente pola máquina virtual durante a evolución do modelo), polo que só se definen métodos de acceso e non de modificación dos seus valores.

6.3.3. Substitución de operadores e variábeis

Para compilar o código de accións e receptividades cun compilador C++ hai que substituír os operadores de evento e temporización, así como os nomes das variábeis almacenadas na máquina virtual durante a execución, polo código que accede aos seus valores utilizando a interface *IVMachineAccess* explicada no apartado anterior. Neste apartado descríbense as modificacións realizadas polo compilador Grafcet no código de accións, condicións de transición e condicións de asociación para obter un código compilábel nun compilador C++.

6.3.3.1. Substitución de eventos

As expresións con eventos poden ser simples, nas que o operador de evento precede ao nome dunha variábel booleana: ↑*bool_var*, ou complexas nas que precede a unha expresión entre parénteses: ↑(*expr*). O seguinte código mostra como se realiza a modificación de expresións que conteñen eventos simples:

```
606. /* Código orixinal (expresión contendo un evento simple) */
607. expr(↑bool_var)
608.
609. /* Código modificado */
610. // declaración e iniciación da variábel auxiliar
611. const bool sfcpp_event_id = vm.getModelVarEvent("bool_var", EDGE_UP);
612.
613. // substitución do evento na expresión pola variábel auxiliar
614. expr(sfcpp_event_id);
```

Cada aparición do evento simple no código é substituído por unha variábel booleana auxiliar: *sfcpp_event_id* (*id* é un identificador numérico único asignado internamente polo compilador). Esta variábel é declarada localmente ao comezo da función na que estea incluído o código (liña 611). O seu valor é iniciado ao resultado devolto polo método

getModelVarEvent[55] cada vez que a función é executada. O método é chamado a través de *vm*, que é unha instancia da clase que implementa a interface *IVMachineAccess* (§6.3.2) pasada como parámetro á función. Nótese que a variábel auxiliar é definida como *const* para evitar que as expresións que poidan modificar o seu valor compilen correctamente. O seguinte código mostra dous exemplos:

```
615. /* Código orixinal */
616. ↑a = true; // asignación errónea dun valor a un evento simple
617.
618. /* Código modificado */
619. const bool sfcpp_event_id = vm.getModelVarEvent("a", EDGE_UP);
620. sfcpp_event_id = true; // ERROR: asignación a unha variábel tipo const
621.
622. /* Código orixinal */
623. any_function(↑a); // función que modifica o parámetro
624.
625. /* Código modificado */
626. const bool sfcpp_event_id = vm.getModelVarEvent("a", EDGE_UP);
627. any_function(sfcpp_event_id); // ERROR: parámetro tipo const
```

A substitución de expresións con eventos complexas realízase simplificando a expresión a unha equivalente na que só se utilicen eventos simples. Cada evento simple é despois substituído utilizando a técnica explicada anteriormente. Para a simplificación das expresións utilízase a aplicación externa Sidoni [146], que implementa o cálculo formal preciso para simplificar as expresións lóxicas permitidas no Grafcet. Sen embargo, comparadas coas expresións C++ estendidas (§6.3.1) as soportadas por Sidoni presentan as seguintes restriccións:

- Unicamente son permitidas expresións booleanas, nas que aparezan os operadores lóxicos: *AND*, *OR* e *NOT*, e os de evento: ↑ e ↓ (Sidoni utiliza os símbolos: ., +, /, > e <, respectivamente).
- As variábeis utilizadas nas expresións son, polo tanto, tamén booleanas e distínguense dous tipos: as entradas de proceso e as variábeis de estado das etapas (de sintaxe X_i).
- Os únicos valores numéricos permitidos son as constantes 0 e 1 (representadas en Sidoni como =0 e =1, respectivamente).

En consecuencia é preciso modificar as expresións con eventos complexas para adaptalas ás características e sintaxe soportada por Sidoni e levar a cabo a súa simplificación. O proceso completo de adaptación e simplificación dunha expresión faise nos pasos seguintes:

1. Substitución dos operadores de temporización por variábeis booleanas auxiliares, tal e como se explica en (§6.3.3.2).
2. Análise sintáctica da expresión para identificar as subexpresións non booleanas.
3. Substitución das expresións non booleanas identificadas por variábeis booleanas. Estas variábeis serán iniciadas ao valor devolto (interpretado como valor booleano) pola subexpresión non booleana á que substitúen. Nótese que en cada subexpresión substituída terá que aplicarse recursivamente o proceso de substitución de variábeis e operadores de evento e temporización.

[55] O método utilizado sería *getSystemVarEvent* no caso de tratarse dunha variábel de proceso.

4. Conversión sintáctica da expresión booleana C++ resultante das transformacións anteriores ao formato soportado por Sidoni. Esta conversión implica substituír os operadores, constantes numéricas e os nomes de variábeis para adaptalos aos utilizados en Sidoni.
5. Simplificación da expresión con Sidoni.
6. Conversión do resultado simplificado da sintaxe Sidoni á sintaxe C++. Esta é a operación contraria á do paso 4.
7. Substitución dos eventos simples e variábeis da expresión simplificada utilizando as técnicas descritas en (§6.3.3.1) e (§6.3.3.3), respectivamente.

O seguinte código mostra un exemplo das diferentes transformacións e substitucións realizadas para simplificar unha expresión complexa:

```
628. /* Código orixinal */
629. if (a && ↑(b || (c+5 > 10) && !T#(10;X_10;20) && (100 || ↓c)))
630.
631. /* Código modificado: paso 1 */
632. // substitución de temporizadores
633. const bool sfcpp_timer_id = vm.getTimerState("sfcpp_timer_id");
634.
635. if (a && ↑(b || (c+5 > 10) && !sfcpp_timer_id && (100 ||  c)))
636.
637. /* Código modificado: pasos 2 e 3 */
638. // identificación e substitución de expresións non booleanas
639. const bool sfcpp_timer_id = vm.getTimerState("sfcpp_timer_id");
640. bool sfcpp_expr_id = (bool) (c+5 > 10);
641.
642. if (a && ↑(b || sfcpp_expr_id && !sfcpp_timer_id && (100 || ↓c)))
643.
644. /* Código modificado: paso 4 */
645. // conversión da expresión ao formato SIDONI
646. // formato C++
647. ↑(b || sfcpp_expr_id && !sfcpp_timer_id && (100 || ↓c))
648.
649. // formato SIDONI
650. >(b + sfcpp_expr_id . /sfcpp_timer_id . (=1 + <c))
651.
652. /* Código modificado: paso 5 */
653. // simplificación da expresión con SIDONI
654. // resultado:
655. (/sfcpp_expr_id . >b) + (sfcpp_timer_id . >b) +
656.    (/sfcpp_timer_id . >sfcpp_expr_id . /b) + (a . <sfcpp_expr_id . /b)
657.
658. /* Código modificado: paso 6 */
659. // conversión da expresión do formato SIDONI a C++
660. const bool sfcpp_timer_id = vm.getTimerState("sfcpp_timer_id");
661. bool sfcpp_expr_id = (bool) (c+5 > 10);
662.
663. if (a && (!sfcpp_expr_id && ↑b) || (sfcpp_timer_id && ↑b) ||
664.    (!sfcpp_timer_id && ↑sfcpp_expr_id && !b) ||
665.    (a && ↓sfcpp_expr_id && !b))
666.
667. /* Código modificado: paso 7 */
668. // substitución de eventos simples
669. const bool sfcpp_timer_id = vm.getTimerState("sfcpp_timer_id");
670. const bool sfcpp_event_id = vm.getSystemVarEvent("b", EDGE_UP);
671. bool sfcpp_expr_id = (bool) (c+5 > 10);
672.
673. if (a && (!sfcpp_expr_id && sfcpp_event_id) ||
674.    (sfcpp_timer_id && sfcpp_event_id) ||
675.    (!sfcpp_timer_id && ↑sfcpp_expr_id && !b) ||
676.    (a && ↓sfcpp_expr_id && !b))
```

No exemplo anterior ponse de relevo que hai que ter en conta unha consideración adicional ao aplicar o paso 7 (liña 667). Como pode comprobarse na expresión resultado da simplificación (liña 673) poden aparecer expresións con eventos simples que afecten ás variábeis auxiliares utilizadas para substituír as expresións non booleanas —*sfcpp_expr_id*, no exemplo—. Estas variábeis son declaradas e iniciadas localmente (liña 671) na función que contén o código e, ao contrario que as variábeis do modelo (entradas, etapas, temporizadores, etc.), o seu valor non é almacenado pola máquina virtual durante a execución, en consecuencia, os seus cambios de estado teñen que ser detectados localmente en cada execución do código da función. O seguinte exemplo mostra como se realizaría a substitución dunha subexpresión non booleana nun caso coma o anterior:

```
677. /* Código orixinal (expresión contendo un evento simple) */
678. ↑expr(subexpr_num)
679.
680. /* Código modificado */
681. // declaración e iniciación das variábeis auxiliares
682. static bool sfcpp_expr_id_last = false;
683. bool sfcpp_expr_id_up, sfcpp_expr_id_down;
684.
685. bool sfcpp_expr_id = (bool) subexpr_num;    // avaliación da subexpresión
686. if (sfcpp_expr_id != sfcpp_expr_id_last)
687. {
688.   sfcpp_expr_id_up = (sfcpp_expr_id_last) ? false : true;
689.   sfcpp_expr_id_down = (sfcpp_expr_id_last) ? true : false;
690.   sfcpp_expr_id_last = sfcpp_expr_id;
691. }
692.
693. // substitución das variábeis auxiliares na expresión
694. expr_simplificada(sfcpp_expr_id[∅|_up|_down]);
```

A técnica utilizada consiste en declarar para cada subexpresión non booleana catro variábeis booleanas: *sfcpp_expr_id* (liña 685), *sfcpp_expr_id_last* (liña 682), *sfcpp_expr_id_up* e *sfcpp_expr_id_down* (liña 683). Estas variábeis almacenan, respectivamente, o valor actual da subexpresión, o valor anterior da subexpresión, a ocorrencia dun cambio no valor da expresión de *false* a *true*, e a ocorrencia dun cambio de *true* a *false*. Na expresión simplificada substitúese a subexpresión e as expresións de evento simple que a afecten polas variábeis auxiliares correspondentes. Nótese que a variábel que almacena o valor anterior é declarada como *static*, de xeito que o seu valor é conservado na memoria entre diferentes execucións da función. Cada vez que esta se executa calcúlase o valor actual da subexpresión (liña 685), se é diferente ao último valor calculado, actualízanse os valores das variábeis que indican a ocorrencia de eventos na subexpresión (liñas 688-690) e, finalmente, execútase a expresión simplificada cos valores calculados para as variábeis auxiliares (liña 694).

Esta técnica ten o inconveniente de introducir variábeis auxiliares innecesarias cando a subexpresión é constante, xa que nese caso o resultado de aplicar un operador de evento á variábel auxiliar é sempre falso. Nesta primeira versión non se implementou o tratamento deste caso específico, deixándose para unha futura optimización do compilador. O seguinte código mostra o resultado do paso 7 do exemplo anterior (liña 667) aplicando esta técnica para substituír a subexpresión non booleana:

```
695. /* Código modificado: paso 7 */
696. // substitución de eventos simples
697. const bool sfcpp_timer_id = vm.getTimerState("sfcpp_timer_id");
698. const bool sfcpp_event_id = vm.getSystemVarEvent("b", EDGE_UP);
699. static bool sfcpp_expr_id_last = false;
700. bool sfcpp_expr_id_up, sfcpp_expr_id_down;
```

```
701. bool sfcpp_expr_id = (bool) (c+5 > 10);
702. if (sfcpp_expr_id != sfcpp_expr_id_last)
703. {
704.   sfcpp_expr_id_up = (sfcpp_expr_id_last) ? false : true;
705.   sfcpp_expr_id_down = (sfcpp_expr_id_last) ? true : false;
706.   sfcpp_expr_id_last = sfcpp_expr_id;
707. }
708.
709. if (a && (!sfcpp_expr_id && sfcpp_event_id) ||
710.     (sfcpp_timer_id && sfcpp_event_id) ||
711.     (!sfcpp_timer_id && sfcpp_expr_id_up && !b) ||
712.     (a && sfcpp_expr_id_down && !b))
```

Nótese que para completar a modificación da expresión con eventos complexa faltaría procesar a subexpresión non booleana (liña 701), substituíndo nela a variábel *c* coa técnica explicada en (§6.3.3.3).

A ubicación do código que calcula os valores das variábeis auxiliares (liñas 704-706) presenta unha complicación adicional, pois é preciso realizar diferentes substitucións dependendo da posición no código na que estea situada a expresión con eventos complexa. Considérense os exemplos seguintes nos que a expresión aparece en diferentes posicións dun bucle *for*:

```
713. /* Exemplo 1: Código orixinal */
714. // expresión na asignación inicial dun for
715. for (bool var = ↑expr(c>5); // c>5, subexpresión non booleana
716.       <any_expr>;
717.       <any_expr>)
718.   c++; // código que modifica c e, polo tanto, o valor da subexpresión
719.
720. /* Código modificado */
721. static bool sfcpp_expr_id_last = false;
722. bool sfcpp_expr_id_up, sfcpp_expr_id_down;
723.
724. bool sfcpp_expr_id = (bool) (c>5)
725. {
726.   sfcpp_expr_id_up = (sfcpp_expr_id_last) ? false : true;
727.   sfcpp_expr_id_down = (sfcpp_expr_id_last) ? true : false;
728.   sfcpp_expr_id_last = sfcpp_expr_id;
729. }
730.
731. for (bool var = expr_simpl(sfcpp_expr_id[Ø|_up|_down]);
732.       <any_expr>;
733.       <any_expr>)
734.   c++;

735. /* Exemplo 2: Código orixinal */
736. // expresión na condición dun for
737. for (<any_expr>; ↑expr(c>5); <any_expr>)
738.   c++;
739.
740. /* Código modificado */
741. static bool sfcpp_expr_id_last = false;
742. bool sfcpp_expr_id_up, sfcpp_expr_id_down;
743.
744. bool sfcpp_expr_id = (bool) (c>5);
745. if (sfcpp_expr_id != sfcpp_expr_id_last)
746. {
747.   sfcpp_expr_id_up = (sfcpp_expr_id_last) ? false : true;
748.   sfcpp_expr_id_down = (sfcpp_expr_id_last) ? true : false;
749.   sfcpp_expr_id_last = sfcpp_expr_id;
750. }
```

```
751. for (<any_expr>; expr_simpl(sfcpp_expr_id[∅|_up|_down]); <any_expr>)
752. {
753.   c++;
754.
755.   sfcpp_expr_id = (bool) (c>5);
756.   if (sfcpp_expr_id != sfcpp_expr_id_last)
757.   {
758.     sfcpp_expr_id_up = (sfcpp_expr_id_last) ? false : true;
759.     sfcpp_expr_id_down = (sfcpp_expr_id_last) ? true : false;
760.     sfcpp_expr_id_last = sfcpp_expr_id;
761.   }
762. }

763. /* Exemplo 3: Código orixinal */
764. // expresión na terceira expresión dun for
765. for (<any_expr>; <any_expr>; ↑expr(c>5))
766.   c++;
767.
768. /* Código modificado */
769. static bool sfcpp_expr_id_last = false;
770. bool sfcpp_expr_id_up, sfcpp_expr_id_down;
771.
772. bool sfcpp_expr_id;
773. for (<any_expr>; <any_expr>; expr_simpl(sfcpp_expr_id[∅|_up|_down]))
774. {
775.   c++;
776.
777.   sfcpp_expr_id = (bool) (subexpr_num);
778.   if (sfcpp_expr_id != sfcpp_expr_id_last)
779.   {
780.     sfcpp_expr_id_up = (sfcpp_expr_id_last) ? false : true;
781.     sfcpp_expr_id_down = (sfcpp_expr_id_last) ? true : false;
782.     sfcpp_expr_id_last = sfcpp_expr_id;
783.   }
784. }
```

Como pode comprobarse nos tres casos mostrados no exemplo anterior, a localización da expresión con eventos complexa no código modifica substancialmente o resultado da substitución. Cando é utilizada na expresión de iniciación da instrucción *for* (liña 715) é abondo con calcular o valor da subexpresión e das variábeis auxiliares antes de iniciarse o bucle. Sen embargo, se a expresión aparece na condición do *for* (liña 737), hai que facer a iniciación antes de iniciarse o bucle e recalcular o valor da expresión e das variábeis auxiliares ao final de cada iteración do bucle. Finalmente, se a expresión está na parte do *for* que se executa despois de cada ciclo (liña 765) non fará falta facer a iniciación antes do bucle, mais si a actualización ao final de cada ciclo. En consecuencia hai que analizar o contexto no que aparece a expresión no código para realizar unha substitución que manteña a súa semántica, especialmente naqueles casos que requiren recalcular o valor das variábeis auxiliares por estar a expresión dentro dun bucle. Os diferentes casos tratados polo compilador son detallados en (§6.5.1.5.3).

6.3.3.2. Substitución de temporizadores

O seguinte código mostra como se realiza a modificación de expresións que conteñen temporizadores:

```
785. /* Código orixinal (expresión contendo un temporizador) */
786. expr(T#(t1;cond;t2))
```

```
787. /* Código modificado */
788. // declaración e iniciación da variábel auxiliar
789. const bool sfcpp_timer_id = vm.getTimerState("sfcpp_timer_id");
790.
791. // substitución do evento na expresión pola variábel auxiliar
792. expr(sfcpp_timer_id);
```

A técnica é a mesma que a explicada para os eventos (§6.3.3.1). Unicamente cambia o nome da variábel auxiliar —*sfcpp_timer_id*— e o método chamado para iniciala —*getTimerState*—. Ademais o compilador ten que almacenar os parámetros do temporizador substituído para compilar con posterioridade a súa condición e xerar a información que permita xestionar o seu estado durante a execución do modelo na máquina virtual.

6.3.3.3. Substitución de variábeis

Na substitución dunha variábel, o código a utilizar depende de se o valor da variábel pode consultarse e modificarse, unicamente consultarse ou unicamente modificarse. É preciso polo tanto considerar os seguintes factores:

1. *O tipo de variábel*, pode tratarse dunha variábel do modelo, do proceso, do estado de activación dunha etapa ou do estado de activación dunha acción.
2. *O código no que se utiliza a variábel*, pode ser nunha condición (o que inclúe as condicións de transición, temporización e as de asociación de acción) ou nunha acción.
3. *O acceso especificado para a variábel*, pode ser de lectura, escritura ou ambas.

Tendo en conta estes factores as seguintes regras definen os diferentes casos a considerar ao realizar a substitución dunha variábel:

1. Os estados de activación de etapas e accións só poden consultarse e non modificarse.
2. As variábeis do modelo definidas polo usuario poden consultarse e modificarse.
3. As variábeis do proceso divídense en dous grupos: as entradas, que só poden consultarse; e as saídas, que só poden modificarse.
4. As condicións non poden producir efectos colaterais, é dicir, as substitucións realizadas no código das condicións teñen que garantir que as variábeis non son modificadas. Esta regra ten prioridade sobre as anteriores.
5. Unha consecuencia das regras 3 e 4 é que as saídas do proceso non poden ser utilizadas nas condicións.

Ademais tamén hai que ter en conta que o método utilizado para acceder ao valor do estado de activación dunha etapa é diferente ao do resto de variábeis (§6.3.2). Os códigos das substitucións realizadas polo compilador nos diferentes casos identificados son os seguintes:

Estado de activación de etapas nas condicións e accións.

```
793. /* Código orixinal */
794. expr(X_id)
795.
796. /* Código modificado */
797. // declaración e iniciación da variábel auxiliar
798. const bool sfcpp_modelvar_id = vm.getStepState(id_num);
799.
800. // substitución da variábel de etapa na expresión pola variábel auxiliar
801. expr(sfcpp_modelvar_id);
```

A técnica é a mesma que a explicada para os eventos (§6.3.3.1) e temporizadores (§6.3.3.2). Unicamente cambia o nome da variábel auxiliar —*sfcpp_modelvar_id*— e o método chamado para iniciala —*getStepState*—. Nótese que neste caso o parámetro pasado a este método non é o nome da variábel senón o identificador numérico único, denominado *id_num*, asignado polo compilador á etapa (*id* e *id_num* son identificadores diferentes). A utilización dunha variábel auxiliar constante garante o cumprimento da regra 1.

Estado de activación de accións, variábeis do modelo e entradas nas condicións e estado de activación de accións nas accións.

```
802. /* Código orixinal */
803. expr(var)
804.
805. /* Código modificado */
806. // declaración e iniciación da variábel auxiliar
807. type sfcpp_modelvar_id_value;
808. vm.getModelVar("var", sfcpp_modelvar_id_value);
809. const type sfcpp_modelvar_id = sfcpp_modelvar_id_value;
810.
811. // substitución da variábel na expresión pola variábel auxiliar
812. expr(sfcpp_modelvar_id);
```

Neste caso o acceso ao valor da variábel faise mediante o método *getModelVar*[56]. Este é un método parametrizado co tipo de dato da variábel accedida (§6.3.2). Este valor é obtido mediante unha variábel auxiliar do tipo do que se trate —*sfcpp_modelvar_id_value* (liña 807)— e asignado a outra variábel do mesmo tipo —*sfcpp_modelvar_id* (liña 809)— que é a utilizada para realizar a substitución. Esta segunda variábel é constante, polo que se garante o cumprimento da regra 4, xa que o seu valor non vai poder ser modificado. A razón de necesitar dúas variábeis do mesmo tipo, unha constante e a outra non, é que ao método *getModelVar* non se lle pode pasar como parámetro unha variábel constante, xa que o seu valor vai ser modificado no interior do método. Sen embargo, para realizar a substitución, e evitar que o valor sexa modificado no código da función, precísase unha variábel auxiliar constante. Nótese tamén que o estado de activación dunha acción é un caso particular das variábeis do modelo, no que o tipo da variábel é booleano e o seu valor non pode ser modificado. A utilización desta técnica cos estados de activación das accións garante tamén o cumprimento da regra 1.

Variábeis do modelo nas accións.

```
813. /* Código orixinal */
814. expr(var)
815.
816. /* Código modificado */
817. // declaración e iniciación da variábel auxiliar
818. pair<type, type> sfcpp_modelvar_id;
819. vm.getModelVar("var", sfcpp_modelvar_id.first);
820. sfcpp_modelvar_id.second = sfcpp_modelvar_id.first;
821.
822. // substitución da variábel na expresión pola variábel auxiliar
823. expr(sfcpp_modelvar_id.second);
824.
825. // actualización do valor da variábel no modelo
826. if (sfcpp_modelvar_id.first != sfcpp_modelvar_id.second)
827.    vm.setModelVar("var", sfcpp_modelvar_id.second);
```

[56] O método utilizado será *getSystemVar* se se trata dunha entrada.

Ao contrario dos casos anteriores, os valores das variábeis do modelo utilizados nas accións si poden ser modificados. Ao realizar a substitución será preciso, polo tanto, incluír non só o código que accede ao valor da variábel senón tamén o que o actualiza en caso de que sexa modificado. Isto é implementado declarando un par de variábeis (liña 818) auxiliares do tipo do que se trate (mediante o "template" *pair* do C++). Os dous membros do par son iniciados ao valor da variábel do modelo (liña 820), que é obtido mediante o método *getModelVar* do mesmo xeito que no caso comentado anteriormente. O segundo membro do par é utilizado para realizar a substitución da variábel do modelo no código. Este membro non é constante, polo que pode ser modificado polo código da acción. A actualización do valor modificado faise nas liñas 826-827, que son inseridas ao final da función que contén o código. Nestas liñas compárase o valor dos dous membros do par, o primeiro conterá o valor da variábel antes de executar o código da acción e o segundo o valor despois de executar o código. Se ambos valores son diferentes actualízase na máquina virtual o valor modificado mediante o método *setModelVar*.

Saídas do proceso nas accións.

Como indica a regra 3, as saídas do proceso poden modificarse mais non consultarse (no caso de precisar a consulta dunha saída, o seu valor e introducido no modelo mediante unha entrada auxiliar). Por este motivo a técnica utilizada para realizar a substitución ten que producir como resultado un código que non compile correctamente nun compilador C++ se a saída é utilizada con calquera outro fin que non sexa o de asignarlle un valor. A modificación realizada é a seguinte:

```
828. /* Código orixinal */
829. expr(output)
830.
831. /* Código modificado */
832. // declaración e iniciación da variábel auxiliar
833. SFCRTOutput<type> sfcpp_modelvar_id;
834.
835. // substitución da variábel na expresión pola variábel auxiliar
836. expr(sfcpp_modelvar_id);
837.
838. // actualización do valor da variábel no modelo
839. if (sfcpp_modelvar_id.modified())
840.   vm.setSystemVar("var", sfcpp_modelvar_id.value());
```

Neste caso a variábel auxiliar utilizada é unha instancia da clase *SFCRTOutput*. Esta é unha clase parametrizábel co tipo de valor da saída. A súa declaración é a seguinte:

```
841. template <class T>
842. class SFCRTOutput
843. {
844.   T output;
845.   bool flag;
846. public:
847.   SFCRTOutput(): flag(false){}
848.   T& operator=(const T& val) { output = val; flag = true; return output; }
849.   bool modified() const { return flag; }
850.   T value() const { return output; }
851. };
```

Cada instancia da clase ten dous atributos, o valor da saída —atributo *output* (liña 844)— e un indicador booleano —atributo *flag* (liña 845)— que indica se ese valor foi modificado con posterioridade á creación da instancia. Inicialmente este indicador ten o valor *false* e só pasa a

valer *true* cando se lle asigna un valor á saída mediante o operador de asignación simple (*operator=*) que é redefinido pola clase (liña 848). A técnica utilizada para a substitución garante que o código resultado produce un erro de compilación se a saída é utilizada nunha expresión que non sexa de asignación. O seguinte código mostra algúns exemplos:

```
852.  /* Exemplo 1: Código orixinal */
853. if (output) // acceso ao valor dunha saída nunha condición
854.
855. /* Código modificado */
856. SFCRTOutput<bool> sfcpp_modelvar_id;
857. if (sfcpp_modelvar_id) // ERROR: expresión ilegal

858. /* Exemplo 2: Código orixinal */
859. int var = output + 5; // acceso ao valor dunha saída nunha asignación
860.
861. /* Código modificado */
862. SFCRTOutput<int> sfcpp_modelvar_id;
863. int var = sfcpp_modelvar_id + 5; // ERROR: expresión ilegal

864. /* Exemplo 3: Código orixinal */
865. output = true; // asignación dun valor a unha saída
866.
867. /* Código modificado */
868. SFCRTOutput<bool> sfcpp_modelvar_id;
869. sfcpp_modelvar_id = true; // OK: asignación correcta
```

En caso de que o valor da saída sexa modificado, o novo valor é actualizado na máquina virtual nas liñas 839-840, que son inseridas ao final da función que contén o código. Nestas liñas comprobase se a saída foi modificada —mediante o método *modified* da clase *SFCRTOutput* (liña 849)—, e en caso afirmativo actualízase na máquina virtual o valor modificado mediante o método *setSystemVar* —o valor modificado é devolto polo método *value* da clase *SFCRTOutput* (liña 850)—.

A técnica descrita ten a limitación de que o valor da saída só pode ser modificado mediante o operador de asignación simple. A asignación de valores á saída utilizando os operadores de asignación compostos[57] do C++ (+=, -=, *=, /=, %=, ^=, &=, |=) ou a utilización da saída como argumento dunha función que modifique o seu valor internamente darían como resultado da substitución un código que provocaría un erro na compilación. Na práctica estas limitacións poden evitarse utilizando variábeis auxiliares. O seguinte código mostra algúns exemplos:

```
870. /* Código orixinal */
871. output = 3;
872. output += 5;  // asignación dun valor a unha saída
873.
874. /* Código modificado */
875. SFCRTOutput<int> sfcpp_modelvar_id;
876. sfcpp_modelvar_id = 3;
877. sfcpp_modelvar_id += 5; // ERROR: expresión ilegal
878.
879. /* Código orixinal corrixido */
880. int aux_var = 3;
881. aux_var += 5;
882. output = aux_var;
883.
```

[57] A asignación en C++ mediante un operador composto require que a variábel modificada teña un valor válido, que será utilizado para calcular o novo valor asignado. Se se utilizaran cunha instancia da clase *SFCRTOutput* antes de que tivese un valor inicial válido, o valor asignado sería erroneo.

```
884. /* Código corrixido modificado */
885. SFCRTOutput<int> sfcpp_modelvar_id;
886. int aux_var = 3;
887. aux_var += 5;
888. sfcpp_modelvar_id = aux_var;   // OK: asignación correcta

889. /* Código orixinal */
890. any_function(output); // función que modifica a saída internamente
891.
892. /* Código modificado */
893. SFCRTOutput<bool> sfcpp_modelvar_id;
894. any_function(sfcpp_modelvar_id); // ERROR: expresión ilegal

895. /* Código orixinal corrixido */
896. bool aux_var;
897. any_function(aux_var);
898. output = aux_var;
899.
900. /* Código corrixido modificado */
901. SFCRTOutput<bool> sfcpp_modelvar_id;
902. bool aux_var;
903. any_function(aux_var);
904. sfcpp_modelvar_id = aux_var;   // OK: asignación correcta
```

6.4. Información para a execución dun modelo Grafcet

O diagrama da Figura 6.8 mostra as clases definidas para representar os modelos Grafcet no formato utilizado durante a súa execución na máquina virtual. Este formato consiste nunha versión compacta do modelo na que se utilizan identificadores numéricos e se inclúe información adicional que permita reducir o número de accesos e avaliacións de condicións durante a execución do modelo. A clase *RTModel*, derivada da clase abstracta *IModule* (§7.1.1.1), agrupa toda a información xerada polo compilador e proporciona unha interface común que permite cargar e almacenar o modelo na máquina virtual. Os detalles sobre como se utiliza esta información durante a execución dos modelos explícanse en (§8.4). Exemplos de diferentes modelos Grafcet representados mediante este formato poden verse en (§6.5.1.2). A descrición detallada das clases mostradas no diagrama da Figura 6.8 é a seguinte[58]:

TimeScale

Enumeración que define as dúas escalas (§3.3.2.4) utilizadas na execución dos modelos Grafcet. Os valores escalares definidos son:

- *ts_internal*, escala interna.
- *ts_external*, escala externa.

RTActionType

Enumeración que define os tipos de accións que se diferencian durante a execución dun modelo Grafcet. Os valores escalares definidos son:

- *model_var_action*, acción que modifica o valor dunha variábel (booleana) do modelo.
- *system_var_action*, acción que modifica o valor dunha variábel (booleana) do proceso.
- *code_block_action*, acción que executa un bloque de código especificado en C++.

[58] Para evitar complicar o diagrama da Figura 6.8 non se mostran as operacións definidas en cada clase, que consisten basicamente en funcións que permiten a inserción e o acceso á información do modelo utilizando os identificadores numéricos dos diferentes elementos.

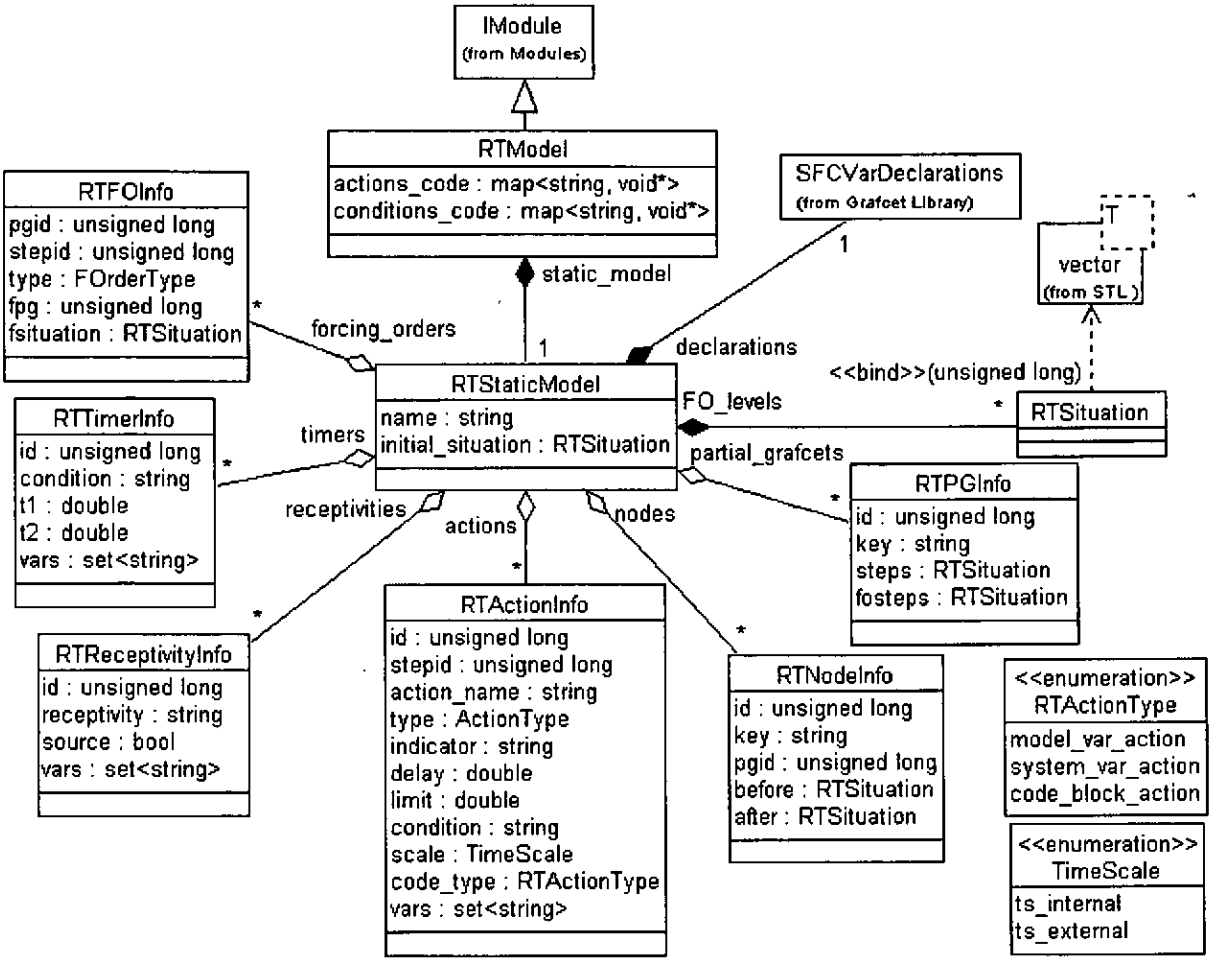


Figura 6.8. Diagrama de clases da información utilizada para a execución dun modelo Grafcet.

RTSituation

Conxunto de identificadores numéricos. Esta clase utilízase tanto para representar a situación do modelo durante a execución como calquera outra información que requira almacenar un conxunto de identificadores de elementos pertencentes ao modelo.

RTModel

Clase que proporciona unha interface común de acceso á información dun modelo Grafcet que sexa cargado na máquina virtual para a súa execución. Os atributos desta clase son os seguintes:

- *actions_code*, información que permite acceder ao código obxecto das accións xerado como resultado da compilación do modelo.
- *conditions_code*, información que permite acceder ao código obxecto das condicións xerado como resultado da compilación do modelo.
- *static_model*, información estructural do modelo.

RTStaticModel

Clase que proporciona unha interface común de acceso á información estática (estructural) do modelo Grafcet durante a execución. Os atributos desta clase son os seguintes[59]:

- *name*, nome do modelo (nome do grafcet global que representa o modelo).
- *initial_situation*, identificadores das etapas que serán activadas ao comezo da execución do modelo.
- *declarations*, declaracións das variábeis utilizadas no modelo.
- *nodes*, información dos nodos contidos no modelo.
- *partial_grafcets*, información dos grafcets parciais do modelo.
- *receptivities*, información das receptividades do modelo.
- *timers*, información dos temporizadores do modelo.
- *actions*, información das accións do modelo.
- *forcing_orders*, información das ordes de forzado do modelo.
- *fo_levels*, información dos niveis da xerarquía de forzado (para cada nivel da xerarquía de forzado almacénanse os identificadores dos grafcets parciais que pertencen ao nivel). Esta información utilízase durante a execución do modelo para aplicar recursivamente as ordes de forzado comezando polos niveis superiores.

RTNodeInfo

Clase que representa a información de cada nodo (etapas e transicións) do modelo Grafcet durante a execución. Os atributos desta clase son os seguintes:

- *id*, identificador interno do nodo (numérico e asignado automaticamente polo compilador).
- *key*, identificador externo do nodo (alfanumérico e asignado polo usuario).
- *pgid*, identificador do grafcet parcial que contén o nodo.
- *before*, identificadores dos nodos que preceden ao nodo na secuencia de control.
- *after*, identificadores dos nodos que suceden ao nodo na secuencia de control.

RTPGInfo

Clase que representa a información de cada grafcet parcial do modelo Grafcet durante a execución. Os atributos desta clase son os seguintes:

- *id*, identificador interno do grafcet parcial (numérico e asignado automaticamente polo compilador).
- *key*, identificador externo do grafcet parcial (alfanumérico e asignado polo usuario).
- *steps*, identificadores das etapas contidas no grafcet parcial.
- *fosteps*, identificadores das etapas contidas no grafcet parcial que teñen asociada algunha orde de forzado.

[59] Os atributos *nodes*, *partial_grafcets*, *receptivities*, *timers*, *actions* e *forcing_orders* foron implementados como mapas de chave numérica (*map<unsigned long, DataClass>*). Isto podería indicarse no diagrama da Figura 6.8 engadindo a chave a cada relación de agregación, mais non se indicou para manter a simplicidade.

RTReceptivityInfo

Clase que representa a información de cada receptividade do modelo Grafcet durante a execución. Os atributos desta clase son os seguintes:

- *id*, identificador da receptividade.
- *receptivity*, nome da función que contén o código da receptividade.
- *source*, atributo que indica se a receptividade está asociada a unha transición fonte.
- *vars*, variábeis (do modelo ou de proceso) utilizadas na receptividade.

RTTimerInfo

Clase que representa a información de cada temporizador do modelo Grafcet durante a execución. Os atributos desta clase son os seguintes:

- *id*, identificador do temporizador.
- *condition*, nome da función que contén o código da condición que activa o temporizador.
- *t1*, cantidade de tempo que a condición debe ser certa para que se active o temporizador.
- *t2*, cantidade de tempo a transcorrer para desactivar o temporizador dende que a condición deixa de ser certa.
- *vars*, variábeis (do modelo ou de proceso) utilizadas na condición do temporizador.

RTActionInfo

Clase que representa a información de cada asociación do modelo Grafcet durante a execución. Os atributos desta clase son os seguintes:

- *id*, identificador da asociación.
- *stepid*, identificador da etapa á que a asociación está asociada.
- *action_name*, este atributo indica o nome dunha variábel do modelo (en accións tipo *model_var_action*) ou de proceso (en accións tipo *system_var_action*) que é modificada durante a activación da asociación. Nas accións tipo *code_block_action* o valor deste atributo indica o nome da función que contén o código da acción executada durante a activación da asociación.
- *type*, tipo de acción (§5.1.2.1).
- *indicator*, nome da variábel do modelo utilizada como indicador da finalización da execución da acción.
- *delay*, cantidade de tempo que indica a demora antes da cal a condición da asociación non é avaliada (a acción non se executa). O valor resultado da avaliación da condición non é tido en conta antes de transcorrer o período de tempo indicado polo valor deste atributo.
- *limit*, cantidade de tempo que indica o límite temporal despois do cal a condición da asociación non pode ser avaliada (a acción non se executa). O valor resultado da avaliación da condición non é tido en conta unha vez transcorrido o período de tempo indicado polo valor deste atributo.
- *condition*, nome da función que contén o código da condición que controla a execución da acción.
- *scale*, atributo que indica si a acción vai ser executada en situacións inestábeis durante as evolucións internas do modelo, ou só nas situacións estábeis.

- *code_type*, o valor deste atributo indica se o atributo *action_name* contén o nome dunha variábel do modelo, de proceso ou o nome dunha acción.
- *vars*, variábeis (do modelo ou de proceso) utilizadas na condición da asociación.

RTFOInfo

Clase que representa a información de cada orde de forzado do modelo Grafcet durante a execución. Os atributos desta clase son os seguintes:

- *pgid*, identificador do grafcet parcial que contén a etapa á que a orde de forzado está asociada.
- *stepid*, identificador da etapa á que a orde de forzado está asociada.
- *type*, tipo de orde de forzado (§5.1.2.1).
- *fpg*, identificador do grafcet parcial forzado.
- *situation*, situación forzada no grafcet parcial indicado polo atributo *fpg*.

6.5. Implementación das fases do compilador Grafcet

No resto desta sección descríbense en detalle as operacións realizadas en cada fase do compilador (§6.2.3), ordenadas dacordo á súa secuencia de execución.

6.5.1. Fase principal ("SFCCompiler")

Esta é a fase principal do compilador, nela crease a estructura de fases e controlase a execución do proceso de compilación utilizando a técnica explicada en (§6.2.4). Ademais nesta fase almacenase a información sobre o modelo Grafcet, as declaracións de variábeis do proceso e as opcións de compilación que se lle pasan ao compilador —atributos *sfc_model*, *systemIO* e *comp_info* da clase *SFCCompiler* (§6.2.1), respectivamente— e iníciase a información sobre directorios e nomes de aplicacións a utilizar durante a compilación —atributo *paths* da clase *SFCCompiler* (§6.2.1)—.

6.5.1.1. Procesamento de macroetapas ("SFCMacroProcessor")

Nesta fase realízanse as seguintes operacións sobre as macroetapas do modelo:

1. Comprobar a corrección sintáctica das macroexpansións. En (§5.1.3.2) definíronse, como parte do metamodelo proposto para o Grafcet, as regras semánticas que teñen que verificar as macroexpansións para que un modelo sexa correcto. A libraría que implementa o metamodelo garante, durante a construcción dun modelo, o cumprimento destas regras coa excepción da que establece que todos os nodos dunha macroexpansión teñen que estar conectados (directa ou indirectamente) entre si. Polo tanto é preciso comprobar o cumprimento desta regra semántica para cada macroexpansión, utilizando para elo a clase auxiliar *SFCMacroChecker* (§5.2.2) incluída na libraría. Esta clase comproba, para unha macroexpansión dada que todos os seus nodos están incluídos no peche transitivo da súa etapa de entrada (o conxunto de nodos directa ou indirectamente conectados a ela).
2. Obter unha versión 'plana' do modelo equivalente ao modelo orixinal. A información sobre a estructura xerárquica que forman os grafcets conexos e as macroetapas non é utilizada durante a execución dun modelo Grafcet. En consecuencia esta información é descartada, substituíndo as macroetapas polas súas macroexpansións e eliminando o nivel intermedio que forman os grafcets conexos entre un grafcet parcial e os nodos que contén. Para elo

utilízase a clase auxiliar *SFCFlattener* (§5.2.2), incluída na libraría que implementa o metamodelo proposto para o Grafcet.

6.5.1.2. Recopilación inicial de información ("SFCInfoHarvester")

Nesta fase realízase a recopilación inicial da información do modelo Grafcet no formato utilizado durante a execución (§6.4) así como a asignación de identificadores numéricos únicos aos diferentes elementos do modelo.

6.5.1.2.1. Recopilación de información

As operacións realizadas para cada elemento do modelo[60] son as seguintes:

Grafcet Global ("SFCGlobal")

1. Almacenar as accións (sen modificalas) no compilador —no atributo *actions_code* da clase *SFCCompiler* (liña 319)—.
2. Procesar os grafcets parciais.

Grafcet Parcial ("SFCPartial")

1. Obter un identificador numérico único.
2. Crear e almacenar —no atributo *pgs* da clase *SFCCompiler* (liña 312)— unha instancia da clase *RTPGInfo* (§6.4) coa información do grafcet parcial.
3. Procesar os nodos (as etapas e transicións) do grafcet parcial.

Nodo ("SFCNode")

1. Obter un identificador numérico único.
2. Crear e almacenar —no atributo *nodes* da clase *SFCCompiler* (liña 313)— unha instancia da clase *RTNodeInfo* (§6.4) coa información do nodo.
3. Representar os arcos orientados substituíndo os apuntadores aos nodos cos que hai unha conexión polos seus identificadores numéricos asignados polo compilador —atributos *before* e *after* da clase *RTNodeInfo* (Figura 6.8)—.
4. Realizar as operacións específicas do tipo de nodo do que se trate (etapa ou transición).

Transición ("SFCTrans")

1. Crear e almacenar —no atributo *recs* da clase *SFCCompiler* (liña 320)— unha instancia da clase *RTReceptivityInfo* (§6.4) coa información da receptividade (o identificador numérico da receptividade é igual ao da transición que a contén).

Etapa ("SFCStep")

1. Se é unha etapa inicial, engadir o seu identificador numérico á situación inicial —no atributo *initial* da clase *SFCCompiler* (liña 311)—.
2. Engadir o identificador numérico á lista de etapas do grafcet parcial que a contén —no atributo *steps* da clase *RTPGInfo* (Figura 6.8)—.
3. Procesar asociacións e ordes de forzado da etapa.

---

[60] Indícase entre parénteses o nome da clase utilizada para representar o elemento na libraría que implementa o metamodelo Grafcet (§5.2).

Asociación ("SFCActionAssociation")

1. Obter un identificador numérico (único para cada asociación).
2. Crear e almacenar —no atributo *actions* da clase *SFCCompiler* (liña 314)— unha instancia da clase *RTActionInfo* (§6.4) coa información da asociación.
3. Se é unha asociación retardada:
   3.1. Obter un identificador numérico (único para cada temporizador).
   3.2. Crear e almacenar —no atributo *timers* da clase *SFCCompiler* (liña 317)— unha instancia da clase *RTTimerInfo* (§6.4) coa información do temporizador (a condición do temporizador é igual a variábel de etapa que contén a asociación).
   3.3. Almacenar na lista de variábeis utilizadas na condición do temporizador o nome da variábel de etapa que contén a asociación —no atributo *vars* da clase *RTTimerInfo* (Figura 6.8)—.
   3.4. Engadir o identificador interno do temporizador (*sfcpp_timer_<id>*) á lista das variábeis utilizadas na condición da asociación —no atributo *vars* da clase *RTActionInfo* (Figura 6.8)—.
   3.5. Engadir o texto '&& sfcpp_timer_<id>' á condición da asociación —no atributo *condition* da clase *RTActionInfo* (Figura 6.8)— para ter en conta o valor do temporizador na avaliación da condición.
4. Se é unha asociación limitada:
   4.1. Obter un identificador numérico (único para cada temporizador).
   4.2. Crear e almacenar —no atributo *timers* da clase *SFCCompiler* (liña 317)— unha instancia da clase *RTTimerInfo* (§6.4) coa información do temporizador (a condición do temporizador é igual a variábel de etapa que contén a asociación).
   4.3. Almacenar na lista de variábeis utilizadas na condición do temporizador o nome da variábel de etapa que contén a asociación —no atributo *vars* da clase *RTTimerInfo* (Figura 6.8)—.
   4.4. Engadir o identificador interno do temporizador (*sfcpp_timer_<id>*) á lista das variábeis utilizadas na condición da asociación —no atributo *vars* da clase *RTActionInfo* (Figura 6.8)—.
   4.5. Engadir o texto '&& !sfcpp_timer_<id>' á condición da asociación —no atributo *condition* da clase *RTActionInfo* (Figura 6.8)— para ter en conta o valor do temporizador na avaliación da condición.

Na Figura 6.9 mostrase un exemplo de como son tratadas as asociacións retardadas e/ou limitadas polo compilador. Para cada restricción temporal defínese un temporizador que ten como condición o estado de activación da etapa que contén a asociación (que ten a forma *X_<id>*, <id> = identificador numérico da etapa). Por exemplo, para a asociación da Figura 6.9.a créanse dous temporizadores: *2s\X_n* para o retardo e *6s\X_n* para o límite. Os identificadores internos destes temporizadores teñen a forma *sfcpp_timer_<id>* (<id> = identificador numérico do temporizador). Para ter en conta os valores destes temporizadores modifícase a condición de activación da asociación, facendo o AND lóxico da condición orixinal, o valor do temporizador de retardo e o valor negado do temporizador de límite (Figura 6.9.b).

Orde de forzado ("SFCForcingOrder")

1. Crear e almacenar —no atributo *forders* da clase *SFCCompiler* (liña 315)— unha instancia da clase *RTFOInfo* (§6.4) (as ordes de forzado non teñen identificador numérico).
2. Almacenar os identificadores numéricos das etapas da situación forzada —no atributo *fsituation* da clase *RTFOInfo* (Figura 6.8)—.
3. Almacenar o identificador numérico do grafcet parcial forzado —no atributo *fpg* da clase *RTFOInfo* (Figura 6.8)—.
4. Engadir o identificador numérico da etapa que contén a orde de forzado á lista de etapas con ordes de forzado no grafcet parcial que contén a etapa —no atributo *fosteps* da clase *RTPGInfo* (Figura 6.8)—.

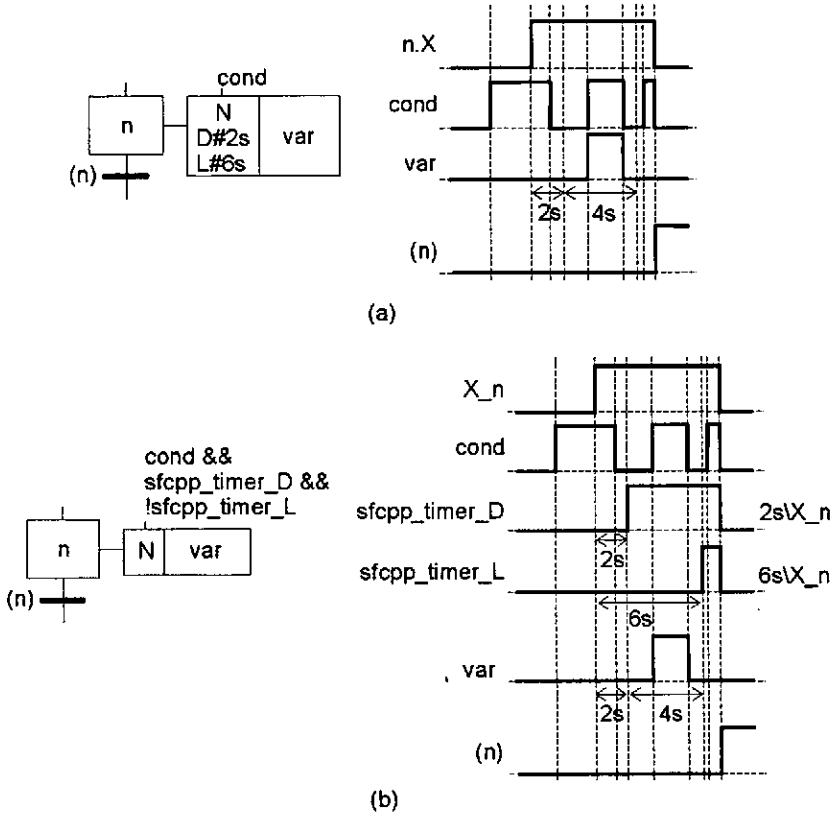


Figura 6.9. Tratamento de accións temporizadas: a) acción retardada e limitada no tempo; e b) modificación realizada polo compilador Grafcet.

6.5.1.2.2. Asignación de identificadores numéricos

A asignación de identificadores numéricos aos elementos do modelo faise do xeito seguinte:

- *Para os grafcets parciais e nodos*, o identificador alfanumérico único proporcionado polo usuario é almacenado nunha cola. O identificador numérico asignado é a posición na cola do identificador alfanumérico.
- *Para as receptividades* utilízase o mesmo identificador que a transición que as contén.
- *Para as asociacións e temporizadores*, o identificador numérico asignado é a posición na cola de asociacións e temporizadores do compilador —nos atributos *actions* e *timers* da clase *SFCCompiler* (liñas 314 e 317, respectivamente)—.
- *Ás ordes de forzado*, non se lles asigna un identificador numérico.

Os identificadores numéricos utilizados teñen un tamaño de catro bytes. O número máximo de elementos do modelo manexados polo compilador son polo tanto:

- Grafcets parciais + etapas + transicións = 4.294.967.296
- Asociacións = temporizadores = 4.294.967.296
- Receptividades = transicións = 4.294.967.296 - (grafcets parciais + etapas)

Na Figura 6.10 e na Figura 6.11 móstranse exemplos da información creada por esta fase para algúns modelos Grafcet concretos.

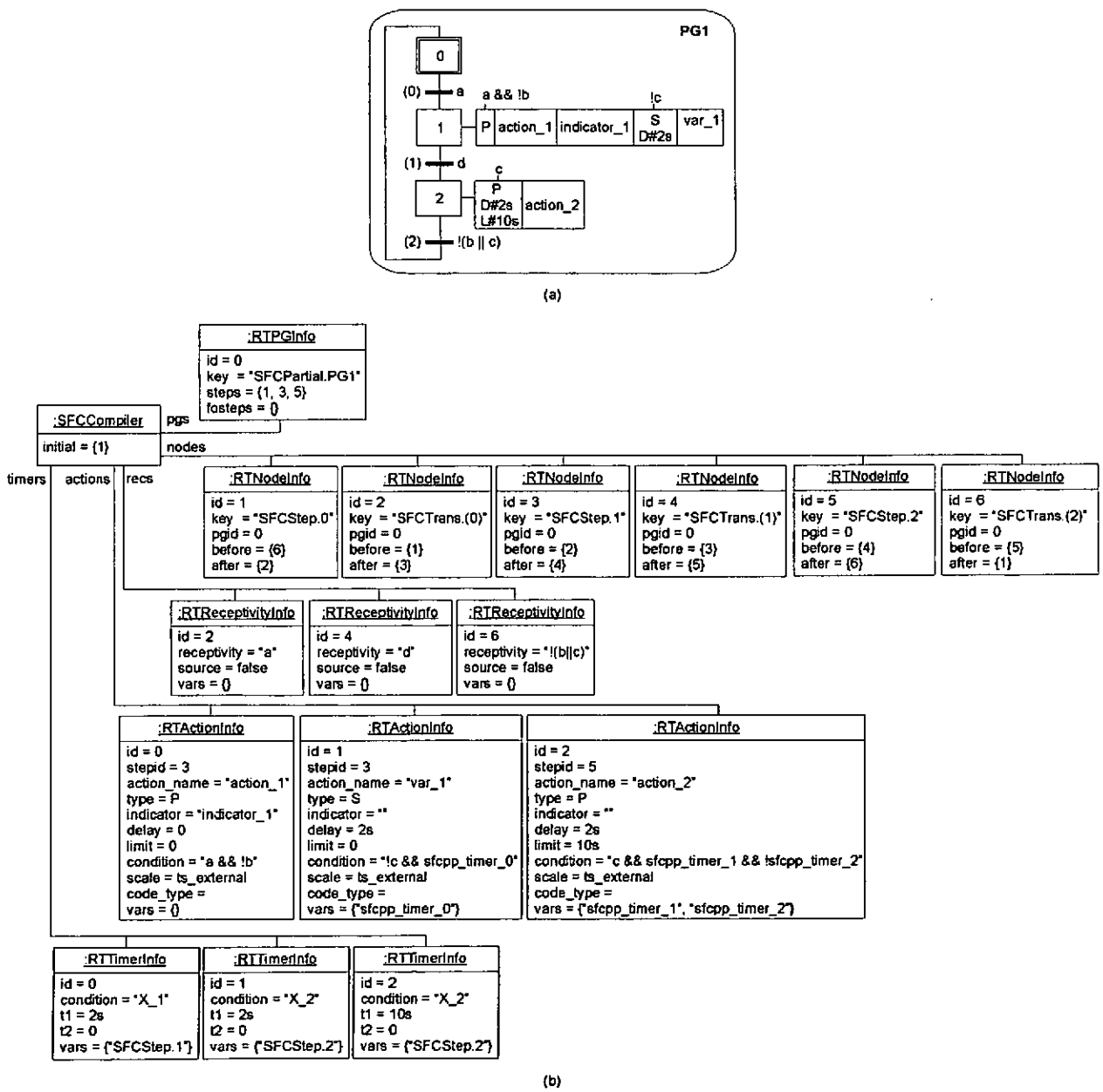
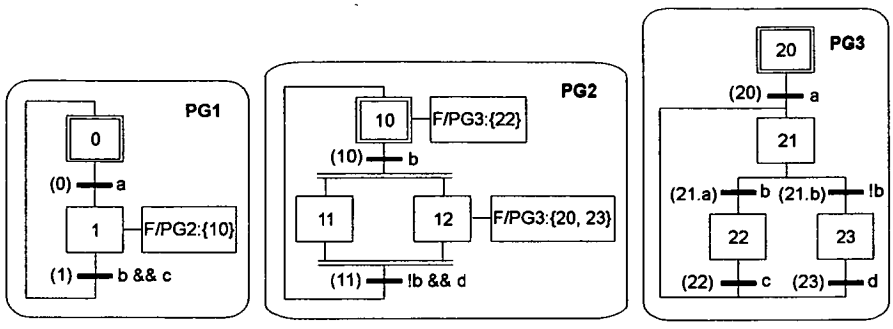
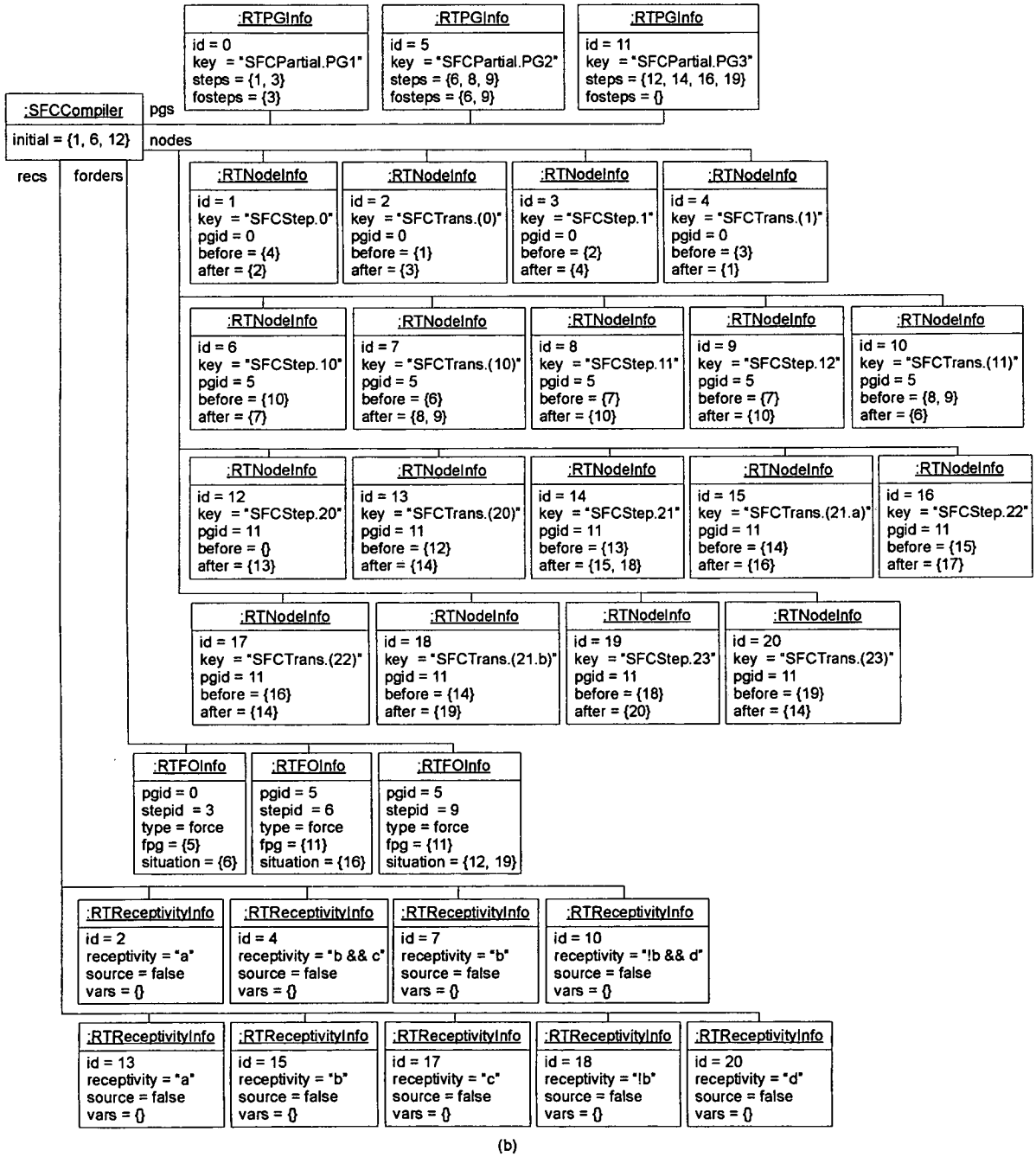


Figura 6.10. Exemplo do funcionamento da fase *SFCInfoHarvester*: a) modelo Grafcet; e b) diagrama de obxectos da información recopilada pola fase.



(a)



(b)

Figura 6.11. Exemplo do funcionamento da fase *SFCInfoHarvester*: a) modelo Grafcet; e b) diagrama de obxectos da información recopilada pola fase.

6.5.1.3. Procesamento das ordes de forzado ("SFCFOrderCompiler")

Nesta fase compróbase a corrección das ordes de forzado do modelo e almacénase a información relativa á xerarquía de forzado. Para cada orde de forzado fanse as comprobacións seguintes:

1. Que o grafcet parcial forzado exista.
2. Que as etapas da situación forzada existan no grafcet parcial forzado.
3. Que a situación forzada sexa baleira se a orde de forzado e de tipo *initial*, *empty* ou *freeze*; e non baleira se é de tipo *force* (§5.1.2.1).

Se todas as ordes de forzado do modelo son correctas, entón comprobase a coherencia da xerarquía de forzado que estas definen. O algoritmo utilizado está baseado no proposto en [103] e é implementado na clase *SFCFOChecker* (§5.2.2) da libraría que implementa o metamodelo Grafcet. Este algoritmo consiste na detección de ciclos no grafo dirixido que forman as ordes de forzado, a xerarquía só será coherente se non contén ciclos. O seguinte pseudocódigo describe o algoritmo utilizado:

```
905. Construír o grafo dirixido que representa a xerarquía de forzado
906. while (algún arco poda ser eliminado)
907.   for each nodo N
908.     calcular número de arcos de entrada (in) e saída (out)
909.     if (in + out > 0 && (in = 0 || out = 0))
910.       eliminar arcos de N do grafo
911.     end if
912.   end for
913. end while
914. if (o grafo non ten arcos)
915.   calcular niveis da xerarquía de forzado
916. end if
```

Basicamente o algoritmo vai eliminando iterativamente os arcos daqueles nodos do grafo que só teñen arcos de entrada ou de saída (son os que non poden formar parte dun ciclo). O grafo non terá ciclos se ao final do proceso iterativo todos os seus arcos foron eliminados. Nese caso calcúlanse e almacénanse —no atributo *fo_levels* da clase *SFCCompiler* (liña 316)— os niveis da xerarquía de forzado. Esta información é utilizada durante a execución do modelo para aplicar iterativamente as ordes de forzado comezando polo nivel máis alto. O algoritmo que calcula os niveis da xerarquía, baseado no proposto en [103], é o seguinte:

```
917. Construír o grafo dirixido que representa a xerarquía de forzado inversa
918. for each nodo N
919.   iniciar o nivel do nodo (nivel := +∞)
920. end for
921. while (algún nodo teña un nivel igual a +∞)
922.   for each nodo N
923.     if (nivel de N = +∞)
924.       nivel de N := 1
925.       for each nodo P predecesor directo de N
926.         if (nivel de P = +∞)
927.           nivel de N := +∞
928.         else
929.           nivel de N := max(nivel de N, nivel de P + 1)
930.         end if
931.       end for
932.     end if
933.   end for
934. end while
```

Este algoritmo comeza construíndo o grafo dirixido que representa a xerarquía de forzado inversa, na que hai un arco dirixido dende cada nodo que represente un grafcet parcial forzado ao nodo que represente o grafcet parcial que o forza. O nivel de cada nodo iníciase a un valor máximo (+∞) utilizado como indicador e posteriormente percórrense iterativamente todos os nodos asignándolles o máximo nivel dos seus predecesores incrementado nunha unidade, sempre e cando ningún dos predecesores teña un nivel igual ao do indicador máximo. Dêste xeito na primeira iteración asígnase o nivel dos nodos sen predecesores (os de nivel 1), na segunda os de nivel 2, e así consecutivamente ata que non quede ningún nodo sen asignárselle nivel.

6.5.1.4. Procesamento das asociacións de acción ("SFCAssociationPreprocessor")

Nesta fase compróbase que as asociacións de acción do modelo (§5.1.2.3) cumpren as seguintes regras:

1. Todas as asociacións teñen que ter un nome.
2. O nome dunha asociación ten que ser unha variábel do modelo, unha saída do proceso ou o nome dunha acción declarada como parte do modelo Grafcet. Ademais ten que ser de tipo booleano e o seu valor modificábel. O tipo de asociación é almacenado no atributo *code_type* da clase *RTActionInfo* (§6.4).
3. O indicador dunha asociación ten que ser unha variábel do modelo ou do proceso declarada como parte do modelo Grafcet.
4. As únicas asociacións que poden definirse como internas son as que, dende un punto de vista externo (§3.3.2.4), teñen —teoricamente— unha duración nula. Polo tanto unicamente as accións impulsionais e almacenadas (P, P0, P1, R ou S) poden ser internas.
5. Pola mesma razón, as asociacións definidas como internas non poden ser retardadas ou limitadas, xa que —teoricamente— non teñen duración na escala de tempo externa.

6.5.1.5. Procesamento do código das accións ("SFCActionCodeCompiler")

Esta fase está composta de varías subfases que procesan o código de cada acción do modelo dacordo ao explicado en (§6.3.3). A execución das subfases configúrase utilizando a técnica explicada en (§6.2.4), e basicamente realizan as operacións seguintes:

1. Preprocesar o código da acción utilizando un preprocesador C++ externo.
2. Substituír os temporizadores utilizando a técnica explicada en (§6.3.3.2).
3. Substituír as variábeis e expresións con eventos, simplificando as expresións con eventos complexas utilizando as técnicas explicadas en (§6.3.3.1) e (§6.3.3.3).

Cada unha das subfases é explicada en detalle a continuación.

6.5.1.5.1. Preprocesamento do código C++ ("CPPActionPreprocessor")

Nesta fase realizase o procesamento das directivas de preprocesador (*#define, #ifdef, #include*, etc.) presentes no código C++. Esta fase é unha clase "wrapper" que capsula o acceso a un preprocesador externo utilizando a técnica explicada en (§6.1.3). Esta clase presupón a existencia dun arquivo denominado *preprocess.bat* que estará almacenado no subdirectorio do directorio *compilers* indicado nas opcións de configuración (§6.1.1).

6.5.1.5.2. Procesamento dos temporizadores ("TimerPreprocessor")

Esta fase recopila e almacena a información dos temporizadores utilizados no código —coa sintaxe explicada en (§6.3.1.1)— e os substitúe por variábeis auxiliares que van almacenar o seu valor durante a execución. Na versión actual do compilador non se permite a utilización de temporizadores no código das condicións de asociación, no código das accións nin o uso de temporizadores aniñados (temporizadores nas condicións doutros temporizadores), polo que na práctica esta fase só se executa cando se compila o código dunha condición de transición. As operacións realizadas por esta fase son as seguintes:

1. Para cada temporizador créase e almacénase —no atributo *timers* da clase *SFCCompiler* (liña 317)— unha instancia da clase *RTTimerInfo* (§6.4) utilizando a información recopilada polo compilador: identificador numérico único asignado automaticamente, código da condición do temporizador e valores de retardo e límite.
2. Substitúense os operadores de temporización polo código que permite acceder ao seu valor durante a execución do modelo aplicando a técnica explicada en (§6.3.3.2). As condicións dos temporizadores son almacenadas para ser procesadas nunha fase posterior do compilador (§6.5.1.8).

6.5.1.5.3. Procesamento de eventos e variábeis ("SFCEventCompiler")

Esta é unha fase composta que realiza a simplificación das expresións con eventos complexas e a substitución das variábeis e eventos simples polo código que permite acceder aos seus valores en tempo de execución. As operacións realizadas por esta fase son as seguintes:

1. Detectar e almacenar a información sobre as expresións con eventos complexas que aparecen no código.
2. Obter a información sobre o contexto (posición no código) no que aparecen estas expresións. Esta información é utilizada posteriormente para substituír as expresións complexas.
3. Reducir e simplificar as expresións con eventos complexas por equivalentes que conteñan unicamente eventos simples dacordo ao explicado en (§6.3.3.1).
4. Obter a información sobre as variábeis do modelo, de proceso, nomes de acción, eventos simples, etc. utilizados no código.
5. Substituír as variábeis e eventos simples utilizando as técnicas explicadas en (§6.3.3).

Cada unha das subfases é detallada a continuación.

Detección das expresións con eventos ("EventExprHarvester")

Esta fase procura as expresións complexas con eventos para reducilas nas fases posteriores. O obxectivo é obter expresións equivalentes que unicamente utilicen eventos simples (cambios de estado en variábeis booleanas), que son os manexados pola máquina virtual. Isto facilita a avaliación de condicións en tempo de execución.

As expresións con eventos complexas son as que comezan por un operador de evento (↑ ou ↓) seguido dunha expresión entre parénteses:

↑(<*expr*>) ou ↓(<*expr*>)

A obrigatoriedade de utilizar os parénteses é unha restricción imposta nesta primeira versión do compilador para facilitar a detección dos límites da expresión. Con esta restricción a implementación desta fase resulta moi simple e non require unha análise sintáctica detallada do

código que sigue ao operador de evento para determinar en que punto remata a expresión. Esta restricción podería eliminarse en futuras implementacións desta fase. Por cada expresión identificada, almacénase a súa posición no código e a expresión contida entre parénteses. Aínda que as expresións con eventos poden aniñarse (unha expresión complexa pode aparecer dentro doutra) isto non é tido en conta nesta fase, na que só son detectadas as expresións máis externas.

Recopilación do contexto das expresións con eventos ("EventExprContextHarvester")

Como foi explicado en (§6.3.3.1), a posición no código na que se utilice unha expresión con eventos complexa pode modificar substancialmente os cambios a realizar no código para simplificala e substituíla polas variábeis auxiliares coas que se implementa a súa semántica. Nesta fase analízase e recopílase a información sobre o contexto no que son utilizadas as expresión con eventos complexas. A parte da gramática C++ que define as instruccións da linguaxe é a seguinte:

```
935. statement:
936.    labeled-statement
937.    expression-statement
938.    compound-statement
939.    selection-statement
940.    iteration-statement
941.    jump-statement
942.    declaration-statement
943.    try-block
944.
945. labeled-statement:
946.    identifier : statement
947.    case constant-expression : statement
948.    default : statement
949.
950. expression-statement:
951.    expression_opt ;
952.
953. compound-statement:
954.    { statement-seq_opt }
955.
956. statement-seq:
957.    statement
958.    statement-seq statement
959.
960. selection-statement:
961.    if ( condition ) statement
962.    if ( condition ) statement else statement
963.    switch ( condition ) statement
964.
965. condition:
966.    expression
967.    type-specifier-seq declarator = assignment-expression
968.
969. iteration-statement:
970.    while ( condition ) statement
971.    do statement while ( expression ) ;
972.    for ( for-init-statement condition_opt ; expression_opt ) statement
973.
974. for-init-statement:
975.    expression-statement
976.    simple-declaration
977.
978. jump-statement:
979.    break ;
980.    continue ;
```

```
981.     return expression_opt ;
982.     goto identifier ;
983.
984.  declaration-statement:
985.     block-declaration
```

Dacordo a esta gramática as expresións con eventos poden ser utilizadas nos contextos seguintes:

1. Na condición dunha instrucción de selección: *if* (liña 961) ou *switch* (liña 963).
2. Na condición dunha instrucción iterativa: *while* (liña 970), *do while* (liña 971) ou *for* (liña 972) —a condición neste caso é opcional—.
3. Na instrucción opcional de iniciación dun bucle *for* (liña 974).
4. Na expresión opcional dun bucle *for* (liña 972), executada ao final de cada ciclo do bucle.
5. Na expresión opcional da instrucción de salto *return* (liña 981).
6. Como parte dunha expresión que por si mesma constitúe unha instrucción (liña 950).
7. Como parte dunha instrucción de declaración (liña 984).

Para realizar a substitución de cada expresión con eventos complexa é preciso determinar en que posición do código orixinal inserir as modificacións. Nos casos mais complicados, nos que a expresión simplificada contén eventos simples aplicados a subexpresións numéricas (§6.3.3.1), poden requirirse modificacións en tres posicións diferentes dependendo do contexto concreto: unha na que inserir a declaración das variábeis auxiliares, outra para a iniciación das subexpresións numéricas e variábeis auxiliares e unha última, cando a expresión orixinal é utilizada na condición dun bucle, para a actualización de subexpresións numéricas e variábeis auxiliares. En consecuencia nesta fase recopílase, para cada expresión con eventos complexa, a información seguinte:

1. Tipo de instrucción (contexto) no que está contida a expresión (p.e. *if*, *for*, etc.).
2. Posición do comezo e fin da instrucción.
3. No caso das instruccións de selección (liña 960) ou iteración (liña 969), que están formadas por unha parte condicional e un conxunto de instruccións executadas cando a condición se cumpre:
   a. A posición do comezo e fin do conxunto de instruccións.
   b. Un indicador booleano de si o conxunto de instruccións está pechado entre '{}'. En caso de non estalo trataríase dunha única instrucción (incluída a baleira: ';').

Os valores desta información para os diferentes contextos diferenciados nesta fase son os seguintes[61]:

Instrucción de selección *if* (tipo = IF)

```
986.  [b]if (<condition_with_complex_event_expression>)
987.    [cb]<statement>; [ce] [e]

988.  [b]if (<condition_with_complex_event_expression>)
989.     { [cb]
990.       <statements>
991.     } [ce] [e]
```

[61] Os valores almacenados pola fase para o inicio e fin da sentencia son indicados coas etiquetas: [b] e [e], e os de inicio e fin do conxunto de instruccións coas etiquétas: [cb] e [ce]. Nas sentencias nas que sexa aplicábel danse duas versións, unha na que o conxunto de instruccións está pechado entre "{}" e outra na que non.

Instrucción de selección *switch* (tipo = SWITCH)

```
992. [b]switch (<condition_with_complex_event_expression>)
993.    {[cb]
994.     case <value_1>: <statements> break;
995.     case <value_n>: <statements> break;
996.     default: <statements> break;
997.    }[ce][e]
```

Instrucción de iteración *for* (3 posicións, tipos = FOR[1|2|3])

```
998. [b]for(<expression_with_complex_event_expression>;
999.        <condition_with_complex_event_expression>;
1000.       <expression_with_complex_event_expression>)
1001.   [cb]<statement>;[ce][e]

1002. [b]for(<expression_with_complex_event_expression>;
1003.       <condition_with_complex_event_expression>;
1004.       <expression_with_complex_event_expression>)
1005.   {[cb]
1006.    <statements>
1007.   }[ce][e]
```

Instrucción de iteración *do while* (tipo = DOWHILE)

```
1008. [b]do
1009.   [cb]<statement>;[ce]
1010.  while (<condition_with_complex_event_expression>);[e]

1011. [b]do
1012.   {[cb]
1013.    <statements>
1014.   }[ce]
1015.  while (<condition_with_complex_event_expression>);[e]
```

Instrucción de iteración *while* (tipo = WHILE)

```
1016. [b]while (<condition_with_complex_event_expression>)
1017.   [cb]<statement>;[ce][e]

1018. [b]while (<condition_with_complex_event_expression>)
1019.   {[cb]
1020.    <statements>
1021.   }[ce][e]
```

Instrucción de salto *return* (tipo = RETURN)

```
1022. [b][cb]return (<condition_with_complex_event_expression>);[ce][e]
```

Calquera outra instrucción (tipo = STATEMENT)

```
1023. [b][cb]<expression_statement_with_complex_event_expression>;[ce][e]
```

As modificacións realizadas e o xeito en que esta información é utilizada para realizalas explícase no apartado seguinte (fase *EventExpressionCompiler*). O pseudocódigo seguinte describe de maneira simplificada o algoritmo utilizado para calcular a información de contexto de cada expresión:

```
1024./* Entradas */
1025.code = código orixinal
1026.code_tam = lonxitude do código orixinal
1027.pos = posición no código orixinal da expresión con eventos complexa
1028.
```

```
1029. /* Saídas */
1030. begin, end = comezo e final da instrucción que contén a expresión
1031. cbegin, cend = comezo e final do código da instrucción
1032. type = tipo de instrucción
1033. code_block = indicador de instrucción simple ou bloque de código
1034.
1035. /* Algoritmo */
1036. // buscar delimitador da instrucción previa
1037. delim_pos = buscar_delimitador_previo(code, pos)
1038. // buscar palabra reservada C++ en code[delim_pos, pos]
1039. stbegin, type = buscar_palabra_reservada(code, delim_pos, pos)
1040. // procesar casos especiais
1041. if (type == STATEMENT)
1042.    // a expresión é parte dun for ou dunha instrucción?
1043.    type = for_ou_expresión(code, delim_pos, pos)
1044. else if (type == WHILE)
1045.    // a expresión é parte dun while ou dun do while?
1046.    type = while_ou_dowhile(code, delim_pos, pos)
1047. end if
1048. if (type == IF or type == WHILE or type == FOR3)
1049.    // está a expresión incluída na condición da instrucción?
1050.    cond_end = buscar_fin_condición(code, stbegin)
1051.    if (pos > cond_end)
1052.       delim_pos = cond_end
1053.       type = STATEMENT
1054.     end if
1055. end if
1056. // calcular información de contexto
1057. begin, end, cbegin, cend, code_block = calcular_contexto(code,
1058.                                                          type, stbegin,
1059.                                                          delim_pos, pos)
```

O algoritmo determina que palabra reservada do C++ hai entre o inicio da instrucción que contén a expresión e a posición da expresión no código. O inicio da instrucción determínase buscando o delimitador da instrucción previa (liña 1037), que pode ser un dos seguintes: '{', '}', ';' ou ':'. Existen dúas situacións nas que algún destes símbolos pode aparecer no código sen ser delimitadores dunha instrucción:

1. O símbolo ';' pode aparecer como separador das expresións dunha instrucción *for* (liña 972). Durante a busca o algoritmo sáltase as expresións entre parénteses, polo que os símbolos ';' que estean na parte da condición dun *for* non serán detectados como delimitadores. A única excepción é cando a expresión con eventos fai parte da condición dun *for*. Este caso explícase posteriormente.
2. O símbolo ':' pode formar parte do operador '?:' (liña 472). Neste caso o algoritmo comproba que o símbolo non fai parte dunha instrucción etiquetada[62] (liña 945) e continúa a busca doutro delimitador.

Unha vez localizado o delimitador búscase entre a súa posición[63] e a posición da expresión no código, unha das seguintes palabras reservadas do C++: *if*, *switch*, *for*, *while* ou *return* (liña 1039). O tipo da instrucción iníciase en función da palabra reservada atopada, en caso de non atopar ningunha o tipo iníciase ao valor *STATEMENT*. Existen algúns casos nos que esta análise inicial non é suficiente para determinar o contexto da expresión e precisan dun procesamento máis específico (liñas 1040-1055). Estes casos son os seguintes:

---

[62] O algoritmo únicamente comproba as instruccións etiquetadas *case* e *default*. A utilización de etiquetas de salto non é habitual e o seu uso non é soportado nesta primeira versión do compilador.

[63] Nótese que no caso de non atopar nengún delimitador isto indicaría que a expresión está contida na primeira das instruccións do código.

Distinción entre unha expresión que é unha instrucción por si propia e unha instrucción for

Cando o delimitador da instrucción previa atopado é ';' e non hai ningunha palabra C++ reservada entre a posición do delimitador[64] e a da expresión con eventos, poden darse dous casos diferentes: que a expresión con eventos estea incluída na parte condicional dunha instrucción *for*, ou nunha expresión que por si propia constitúe unha instrucción. Os seguintes exemplos mostran as diferentes posibilidades:

```
1060. /* caso 1: a expresión fai parte dun for */
1061. // 1.a: condición do for (tipo = FOR2)
1062. for(<for_initial_expression>;[delim_pos]
1063.     [st_begin]<condition_with_complex_event_expression>;
1064.     <expression>)
1065.  <statement>;
1066.
1067. // 1.b: expresión do for executada ao final de cada ciclo (tipo = FOR3)
1068. for(<for_initial_expression>;
1069.     <condition>;[delim_pos]
1070.     [st_begin]<expression_with_complex_event_expression>)
1071.   <statement>;
1072.
1073. /* caso 2: a expresión non fai parte dun for */
1074. <statement>;[delim_pos]
1075. [st_begin]<statement_with_complex_event_expression>;
```

O algoritmo identifica as diferentes posibilidades e corrixe axeitadamente o valor de inicio da instrucción[65] e o do tipo de contexto, que pasará a ser *FOR2*, *FOR3* ou *STATEMENT* dependendo do caso concreto.

Distinción entre unha instrucción while e unha do while

Se a palabra C++ reservada atopada entre a posición do delimitador da instrucción previa e a da expresión con eventos é *while*, poden darse dous casos: que a expresión con eventos estea incluída nunha instrucción *while* ou na parte condicional dunha instrucción *do while*. Os seguintes exemplos mostran as diferentes posibilidades:

```
1076. /* caso 1: a expresión fai parte dun while */
1077.          <statement>;[delim_pos]
1078. [st_begin]while(<condition_with_complex_event_expression>)
1079.             <statement>;
1080.
1081. /* caso 2: a expresión fai parte dun do while */
1082.          do
1083.            <statement>;[delim_pos]
1084. [st_begin]while(<condition_with_complex_event_expression>);
```

O algoritmo distingue entre ambas e no caso de tratarse dunha instrucción *do while* corrixe o valor de inicio da instrucción e o do tipo de contexto, que pasará a ser *DOWHILE*. A determinación do inicio da instrucción *do while* require buscar a palabra reservada *do* que se corresponda co *while* atopado e que apareza no código nunha posición anterior á súa, tendo en conta que as instruccións *do while* poden aniñarse entre si e con outras instruccións *while*, tal e como mostra o exemplo seguinte:

---

[64] Indicada como [*delim_pos*].
[65] Indicado como [*st_begin*].

```
1085.          do // inicio da instrucción do while
1086.           while(<condition>)
1087.            do
1088.            {
1089.             if (<condition>)
1090.              do
1091.               do
1092.                while(<condition>); // while cunha instrucción baleira
1093.               while(<condition>);
1094.              while(<condition>);
1095.             else
1096.              do ; // do while cunha instrucción baleira
1097.              while(<condition>);
1098.            }
1099.           while(<condition>);[delim_pos]
1100. [st_begin]while(<condition_with_complex_event_expression>);
```

Distinción entre unha expresión utilizada na condición ou na primeira instrucción do código dunha instrucción if, while ou for

Se a palabra C++ reservada atopada é *if*, *while* ou *for* é preciso distinguir se a expresión é parte da condición da instrucción ou da primeira das instruccións do seu código. O seguinte exemplo mostra as dúas posibilidades para unha instrucción *if*:

```
1101./* caso 1: a expresión é parte da condición */
1102.          <statement>;[delim_pos]
1103.[st_begin]if (<condition_with_complex_event_expression>)[cond_end]
1104.            <statement>;
1105.
1106./* caso 2: a expresión é parte da primeira instrucción */
1107.          <statement>;[delim_pos]
1108.[st_begin]if (<condition>)[cond_end]
1109.            <statement_with_complex_event_expression>;
```

Neste exemplo o algoritmo detectaría a palabra reservada *if* entre a posición do delimitador da instrucción previa e a posición da expresión no código. O tipo da instrucción iniciaríase ao valor *IF* e a posición de inicio iniciaríase á posición do *if* no código. Sen embargo, no segundo caso mostrado no exemplo é preciso corrixir o tipo e posición de inicio da instrucción (liñas 1106-1109). O tipo pasará a ser *STATEMENT* e a posición de inicio será igual ao final da condición[66]. Este mesmo razoamento é aplicábel ás instruccións *while* e *for*. Nas instruccións *switch* e *do while* non aparece este problema xa que a súa condición sempre vai seguida dos delimitadores '{' e ';', respectivamente. Polo tanto sempre que se detecte as palabras reservadas *switch* ou *while* nunha destas instruccións, a única posibilidade é que a expresión estea incluída na condición da instrucción.

Unha vez identificado o tipo de instrucción no que está incluída a expresión, o último paso do algoritmo (liña 1057) consiste na recopilación da información de contexto explicada anteriormente: posicións de inicio e fin da instrucción, posicións de inicio e fin do código da instrucción e determinación de se o código está pechado entre '{}'. A parte do algoritmo que realiza este cálculo ten en conta as consideracións seguintes:

1. Cando a expresión está incluída na condición dunha instrucción *if*, hai que determinar se é o *if* inicial ou é parte dunha póla *else if*. O seguinte código mostra un exemplo:

```
1110. /* caso 1: a expresión está na condición dun if */
1111.[begin]if (<condition_with_complex_event_expression>)
1112.          <statement_seq>
```

[66] Indicado como [*cond_end*].

```
1113. /* caso 2: a expresión está na condición dun else if */
1114. [begin]if (<condition>)
1115.          <statement_seq>
1116.       else if (<condition_with_complex_event_expression>)
1117.          <statement_seq>
```

Polo tanto a localización do inicio da instrucción require buscar a palabra reservada *if* que non vaia antecedida dun *else* e que se corresponda co *else if* no que está a expresión. Do mesmo xeito que acontecía nas instruccións *do while* explicadas anteriormente o algoritmo ten en conta que as instruccións *if* poden aniñarse entre si e con outras instruccións, tal e como mostra o exemplo seguinte:

```
1118. [begin]if (<condition>) // inicio da instrucción if
1119.          if (<condition>)
1120.             if (<condition>)
1121.                <statement_seq>
1122.             else
1123.                <statement_seq>
1124.          else
1125.             <statement_seq>
1126.       else if (<condition>)
1127.          if (<condition>)
1128.             <statement_seq>
1129.          else
1130.             <statement_seq>
1131.       else if(<condition_with_complex_event_expression>)
1132.          <statement_seq>
```

2. A localización do final do código dunha instrucción pode implicar unha busca recursiva cando este está formado por múltiples instruccións aniñadas unhas dentro doutras. O seguinte código mostra un exemplo:

```
1133.while (<condition_with_complex_event_expression>)
1134.   if (<condition>)
1135.      <statement_seq>
1136.   else
1137.      for (<initial_for_expr>;<condition>;<expression>)
1138.        <statement>; [cend]
```

Neste exemplo, a determinación do final do código da instrucción *while*[67] faise buscando o punto no que remata a instrucción *for* contida na parte *else* da instrucción *if*.

En resume, a implementación desta fase ten en conta todas as consideracións indicadas para a obtención da información de contexto dunha expresión con eventos complexa. Os detalles das partes do algoritmo que permiten diferenciar os diferentes contextos e calcular a súa información non foron incluídos para evitar complicar en exceso a explicación desta fase.

Simplificación de expresións con eventos complexas ("EventExpressionCompiler")

Esta fase encárgase de simplificar as expresións con eventos complexas calculando unha equivalente que só utilice eventos simples, dacordo ao explicado en (§6.3.3.1). Para a reducción das expresións utilízase a aplicación externa Sidoni (§1.3.3), que unicamente reduce expresións lóxicas con eventos, polo que é preciso adaptar as expresións C++ (que poden conter subexpresións numéricas) para que sexan aceptadas por Sidoni. A adaptación de cada expresión detectada na fase *EventExprHarvester* realízase mediante as operacións seguintes:

[67] Indicado como [*cend*].

1. Analízase sintacticamente a expresión, obtendo como resultado a árbore da sintaxe abstracta (AST) que a representa. Esta AST é utilizada para identificar as subexpresións numéricas e as subexpresións con eventos complexas aniñadas.
2. Modifícase a AST da expresión, substituíndo as subexpresións numéricas e as subexpresións con eventos complexas aniñadas por variábeis booleanas. No código resultado da modificación, estas variábeis serán iniciadas ao valor devolto pola subexpresión á que substitúen.
3. Simplifícase a AST modificada utilizando Sidoni, aplicación que utiliza as regras do álxebra de Boole estendidas (que teñen en conta os flancos no valor das variábeis booleanas) para simplificar expresións lóxicas.
4. Procésanse as expresións con eventos complexas aniñadas, aplicando as operacións anteriores a cada unha delas.
5. Modifícase o código orixinal, substituíndo a expresión inicial pola equivalente simplificada e engadindo o código para o manexo das expresións non booleanas segundo o explicado en (§6.3.3.1).

Cada un destes pasos é explicado en detalle a continuación.

Análise sintáctica da expresión, obtención da AST

Para realizar a análise sintáctica da expresión implementouse un analizador sintáctico tomando como base a gramática[68] descrita en (§6.3.1.2). O analizador foi xerado automaticamente mediante a aplicación ANTLR, que é unha ferramenta que permite xerar automaticamente analizadores sintácticos pred-LL(k): analizadores descendentes recursivos con k "tokens" de adianto e soporte á utilización de predicados semánticos e sintácticos para a resolución de ambigüidades na gramática. Esta utilidade é parte do PCCTS (§1.3.4), un conxunto de ferramentas para a xeración automática de compiladores [135].

A definición da gramática no formato utilizado por ANTLR pode verse no Anexo D. Esta gramática está anotada con símbolos e accións que dirixen o proceso de construcción da AST de cada expresión. O acceso ao analizador sintáctico faise mediante unha clase auxiliar denominada *CPPParserWrapper*, cuxa interface pública é a seguinte:

```
1139. class CPPParserWrapper
1140. {
1141.   void operator()(const string& input, ASTBase **ast);
1142. };
```

Os detalles do código que executa as aplicacións auxiliares xeradas polo PCCTS para realizar a análise sintáctica das expresións son ocultados na implementación do operador *operator()*. Este método recibe o código da expresión a analizar —no parámetro *input*— e devolve a AST da expresión —no parámetro *ast*—. Os seguintes exemplos mostran as ASTs xeradas para algunhas expresións:

[68] Coa excepción dos temporizadores que son substituidos previamente na fase *TimerPreprocessor*, o que permite restrinxir o seu uso únicamente ás condicións de transición.

```
1143. ↑(a && ↓(b || 3+c > 5 || ↑(a > b+3)) && !fun(ob->sys.dat[i++]))
1144.
1145. ↑
1146. |
1147. ( - )
1148. |
1149. && — a
1150. |
1151. && ———————— ↓
1152. |              |
1153. !              ( - )
1154. |              |
1155. ( - fun        || ———————— ↑
1156. |              |           |
1157. [ - ++ - i     || - b      ( - )
1158. |              |           |
1159. . - dat        > - 5       > - a
1160. |              |           |
1161. -> - ob - sys  + - 3 - c   + - b - 3
```

```
1162. ↑(500+class::attrib > ((↑(c > d*3 || ↓(c || d/3 > b)))?12:(26*b)) || ↓c)
1163.
1164. ↑
1165. |
1166. ( - )
1167. |
1168. || ——————————————————————————————— ↓
1169. |                                     |
1170. > ———————————————————— ( - )          c
1171. |                      |
1172. + - 500 - class::attrib ? - 12 - : - ( - )
1173.                        |           |
1174.                        ( - )       * - 26 - b
1175.                        |
1176.                        ↑
1177.                        |
1178.                        ( - )
1179.                        |
1180.                        || ———————— ↓
1181.                        |           |
1182.                        > - c       ( - )
1183.                            |       |
1184.                            * - d - 3   || - c
1185.                                        |
1186.                                        > - b
1187.                                        |
1188.                                        / - d - 3
```

Reducción da AST da expresión

Para realizar a modificación da AST substituíndo as subexpresións numéricas e as subexpresións con eventos complexas por variábeis auxiliares booleanas utilizouse un analizador\traductor de árbores sintácticas xerado automaticamente mediante a aplicación Sorcerer, que é outra das ferramentas do PCCTS. A gramática utilizada para xerar o analizador pode verse no Anexo E. O acceso a este analizador faise mediante unha clase auxiliar denominada *CPPTreeParser*, cuxa interface pública é a seguinte:

```
1189. class CPPTreeParser
1190. {
1191.   void parse(SORASTBase **root, SORASTBase **result,
1192.              astdata_seq& expressions, unsigned& init_number);
1193. };
```

O método *parse* implementa o proceso de análise e substitución da AST da expresión. Este método recibe dous argumentos: a AST orixinal —no parámetro *root*— e un índice utilizado para numerar as variábeis auxiliares nas substitucións —no parámetro *init_number*—; e devolve a AST da expresión resultado —no parámetro *result*— e as ASTs das subexpresións substituídas —no parámetro *expressions*—. Os seguintes exemplos mostran os resultados obtidos ao reducir as expresións das liñas 1143 e 1162:

```
1194. /* Exemplo 1: Expresión orixinal */
1195. ↑(a && ↓(b || 3+c > 5 || ↑(a > b+3)) && !fun(ob->sys.dat[i++]))
1196.
1197. /* Resultado */
1198. ↑(a && ↓(b || sfcpp_expr_1 || ↑(sfcpp_expr_2)) && !sfcpp_expr_3)
1199.
1200. /* Subexpresións substituídas */
1201. sfcpp_expr_1 = 3+c > 5
1202. sfcpp_expr_2 = a > b+3
1203. sfcpp_expr_3 = fun(ob->sys.dat[i++])

1204. /* Exemplo 2: Expresión orixinal */
1205. ↑(500+class::attrib > ((↑(c > d*3 || ↓(c || d/3 > b)))?12:(26*b)) || ↓c)
1206.
1207. /* Resultado */
1208. ↑(sfcpp_expr_1 || ↓c)
1209.
1210. /* Subexpresións substituídas */
1211. sfcpp_expr_1 = 500+class::attrib > ((sfcpp_expr_2)?12:(26*b))
1212.
1213. /* Subexpresións aniñadas */
1214. sfcpp_expr_2 = ↑(sfcpp_expr_3 || sfcpp_expr_4)
1215. sfcpp_expr_3 = c > d*3
1216. sfcpp_expr_4 = ↓(c || sfcpp_expr_5)
1217. sfcpp_expr_5 = d/3 > b
```

No caso da primeira expresión (liña 1195) o analizador detecta tres subexpresións numéricas (non booleanas), que substitúe por variábeis booleanas auxiliares (coa sintaxe *sfcpp_expr_id*). A segunda das expresións (liña 1205) contén unha única subexpresión numérica que é substituída polo analizador. Esta subexpresión —*sfcpp_expr_1*— contén á súa vez unha expresión con eventos complexa (a condición do operador '?:'), polo que o proceso de reducción é aplicado recursivamente, e esta subexpresión é substituída pola variábel *sfcpp_expr_2*. O proceso continua aplicándose recursivamente ás subexpresións numéricas e con eventos contidas en *sfcpp_expr_2* ata que non queda ningunha sen substituír.

Simplificación da expresión

Unha vez reducida a expresión orixinal para que conteña unicamente subexpresións booleanas, a simplificación da expresión faise utilizando a aplicación externa Sidoni [146], dacordo ao explicado en (§6.3.3.1). Esta aplicación utiliza as regras do álxebra de Boole estendidas co soporte á utilización de eventos para simplificar expresións lóxicas. O acceso a esta aplicación faise mediante a clase auxiliar *SidoniWrapper*, que se encarga de converter unha expresión booleana C++ ao formato utilizado por Sidoni, executar a aplicación —utilizando a técnica explicada en (§6.1.3)— e converter o resultado de novo á sintaxe do C++. A Táboa 6-II mostra a correspondencia entre as características soportadas por Sidoni e o equivalente C++ estendido cos operadores de evento e temporización.

Característica	Sidoni	C++
Operadores booleanos	AND = . OR = + NOT = /	AND = && OR = || NOT = !
Operadores con eventos	> e <	↑ e ↓
Constantes numéricas	=0 =1	Calquera valor numérico
Variábeis	X_{id} = identificador de etapa id = variábel de proceso	$X_{\_id}$ = identificador de etapa sfcpp_timer_id = temporizador sfcpp_expr_id = variábel auxiliar id = variábel de proceso, do modelo ou identificador de acción

Táboa 6-II. Equivalencia entre os operadores Sidoni e os operadores C++.

O exemplos seguintes mostran o resultado da simplificación das expresións das liñas 1143 e 1162, representadas no formato Sidoni e na sintaxe C++ (nótese que no caso da expresión da liña 1162, tamén son simplificadas as subexpresións con eventos complexas aniñadas):

```
1218. /* Exemplo 1: Expresión orixinal */
1219. ↑(a && ↓(b || 3+c > 5 || ↑(a > b+3)) && !fun(ob->sys.dat[i++]))
1220.
1221. /* Expresión reducida */
1222. // Sintaxe C++
1223. ↑(a && ↓(b || sfcpp_expr_1 || ↑(sfcpp_expr_2)) && !sfcpp_expr_3)
1224. // Sintaxe Sidoni
1225. >(a . <(b + sfcpp_expr_1 + >(sfcpp_expr_2)) . /sfcpp_expr_3)
1226.
1227. /* Expresión simplificada */
1228. // Sintaxe Sidoni
1229. (<b . /sfcpp_expr_1 . a . /sfcpp_expr_3) +
1230.    (<sfcpp_expr_3 . /b . a . /sfcpp_expr_3)
1231. // Sintaxe C++
1232. (↓b && !sfcpp_expr_1 && a && !sfcpp_expr_3) ||
1233.    (↓sfcpp_expr_3 && !b && a && !sfcpp_expr_3)

1234. /* Exemplo 2: Expresión orixinal */
1235. ↑(500+class::attrib > ((↑(c > d*3 || ↓(c || d/3 > b)))?12:(26*b)) || ↓c)
1236.
1237. /* Expresión reducida */
1238. // Sintaxe C++
1239. ↑(sfcpp_expr_1 || ↓c)
1240. // Sintaxe Sidoni
1241. >(sfcpp_expr_1 + <c)
1242.
1243. /* Expresión simplificada */
1244. // Sintaxe Sidoni
1245. >sfcpp_expr_1 + (<c . /sfcpp_expr_1)
1246. // Sintaxe C++
1247. ↑sfcpp_expr_1 || (↓c && !sfcpp_expr_1)
1248.
1249. /* Subexpresións con eventos aniñadas reducidas */
1250. // Sintaxe C++
1251. ↑(sfcpp_expr_3 || sfcpp_expr_4)     // sfcpp_expr_2
1252. ↓(c || sfcpp_expr_5)               // sfcpp_expr_4
```

```
1253. // Sintaxe Sidoni
1254. >(sfcpp_expr_3 + sfcpp_expr_4)      // sfcpp_expr_2
1255. <(c + sfcpp_expr_5)                 // sfcpp_expr_4
1256. /* Subexpresións con eventos aniñadas simplificadas */
1257. // Sintaxe Sidoni
1258. (/sfcpp_expr_4 . >sfcpp_expr_3) +              // sfcpp_expr_2
1259.    (>sfcpp_expr_4 . /sfcpp_expr_3)
1260. (<c . /sfcpp_expr_5) + (<sfcpp_expr_5 . /c)    // sfcpp_expr_4
1261. // Sintaxe C++
1262. (!sfcpp_expr_4 && ↑sfcpp_expr_3) ||            // sfcpp_expr_2
1263.    (↑sfcpp_expr_4 && !sfcpp_expr_3)
1264. (↓c && !sfcpp_expr_5) || (↓sfcpp_expr_5 && !c)  // sfcpp_expr_4
```

Hai un aspecto adicional a considerar na utilización de Sidoni. Esta aplicación ten en conta automaticamente as seguintes hipóteses do modelo Grafcet (§3.3.2.1):

1. Non é posíbel a ocorrencia simultánea de dous eventos externos non relacionados ($\uparrow a$ *and* $\uparrow b = 0$).
2. Non é posíbel a ocorrencia simultánea dun evento externo e un interno ($\uparrow a$ *and* $\uparrow X_i = 0$).

Sen embargo na implementación do algoritmo de interpretación dos modelos Grafcet utilizado na máquina virtual (§8.3.1.3) esta é unha característica configurábel, que pode aplicarse ou non dacordo ao indicado polo usuario. Polo tanto é preciso evitar que Sidoni aplique automaticamente estas hipóteses na simplificación de expresións, para que o resultado da simplificación poda ser utilizado nambos casos. A técnica utilizada consistiu en preceder todas as variábeis utilizadas na expresión orixinal coa letra *X*, de modo que Sidoni interpreta que todas as variábeis utilizadas son internas (variábeis de etapa) e non se dan, polo tanto, as condicións para a aplicación das hipóteses indicadas. Unha vez simplificada a expresión elimínanse os prefixos das variábeis. En certos casos o resultado obtido non é a simplificación mínima que podería obterse, mais conserva a semántica da expresión orixinal durante a execución dos modelos tanto se se teñen en conta as hipóteses anteriores coma se non.

Modificación do código

A última operación realizada por esta fase do compilador consiste na modificación do código da expresión orixinal polo código da expresión simplificada, así como a inserción do código que declara, inicia e actualiza as variábeis auxiliares utilizadas para a substitución das expresións numéricas e con eventos aniñadas, dacordo á técnica explicada en (§6.3.3.1). O código modificado a utilizar depende do contexto da expresión, tal e como se indica en (§6.3.3.1). As diferentes posibilidades tratadas polo compilador e a localización do código auxiliar en cada caso son as seguintes:

Instrucción de selección *if* (tipo = IF)

```
1265. [declaración + iniciación]
1266. if (<condition_with_complex_event_expression>)
1267.   <statement>;
```

Instrucción de selección *switch* (tipo = SWITCH)

```
1268. [declaración + iniciación]
1269. switch (<condition_with_complex_event_expression>)
1270. {
1271.   case <value_1>: <statements> break;
1272.   case <value_n>: <statements> break;
1273.   default: <statements> break;
1274. }
```

Instrucción de iteración *for* (3 posicións, tipo = FOR[1|2|3])

```
1275. // FOR1
1276. [declaración + iniciación]
1277. for(<expression_with_complex_event_expression>;
1278.     <condition>;
1279.     <expression>)
1280.  <statement>;
1281.
1282. // FOR2
1283. [declaración + iniciación]
1284. for(<expression>;
1285.     <condition_with_complex_event_expression>;
1286.     <expression>)
1287. {
1288.    <statements>
1289.    [actualización]
1290. }
1291.
1292. // FOR3
1293. [declaración]
1294. for(<expression>;
1295.     <condition>;
1296.     <expression_with_complex_event_expression>)
1297. {
1298.    <statements>
1299.    [actualización]
1300. }
```

Instrucción de iteración *do while* (tipo = DOWHILE)

```
1301. [declaración]
1302. do
1303. {
1304.  <statements>
1305.  [actualización]
1306. }
1307. while (<condition_with_complex_event_expression>);
```

Instrucción de iteración *while* (tipo = WHILE)

```
1308. [declaración + iniciación]
1309. while (<condition_with_complex_event_expression>)
1310. {
1311.  <statements>
1312.  [actualización]
1313. }
```

Instrucción de salto *return* (tipo = RETURN)

```
1314. [declaración + iniciación]
1315. return (<condition_with_complex_event_expression>);
```

Calquera outra instrucción (tipo = STATEMENT)

```
1316. [declaración + iniciación]
1317. <expression_statement_with_complex_event_expression>;
```

Os seguintes exemplos mostran o código xerado para a substitución das expresións das liñas 1143 e 1162 cando son utilizadas en contextos tipo *FOR2* e *DOWHILE*, respectivamente:

```
1318./* Código orixinal (FOR2) */
1319.for(<expression>;
1320.    ↑(a && ↓(b || 3+c > 5 || ↑(a > b+3)) && !fun(ob->sys.dat[i++]));
1321.    <expression>)
1322.{
1323.   <statements>
1324.}
1325.
1326./* Código modificado */
1327.// declaración + iniciación variábeis auxiliares
1328.bool sfcpp_expr_1 = (bool) (3+c > 5);
1329.bool sfcpp_expr_2 = (bool) (a > b+3);
1330.bool sfcpp_expr_3 = (bool) (fun(ob->sys.dat[i++]));
1331.for(<expression>;
1332.    (↓b && !sfcpp_expr_1 && a && !sfcpp_expr_3) ||
1333.       (↓sfcpp_expr_3 && !b && a && !sfcpp_expr_3);
1334.    <expression>)
1335.{
1336.   <statements>
1337.   // actualización variábeis auxiliares
1338.   sfcpp_expr_1 = (bool) (3+c > 5);
1339.   sfcpp_expr_2 = (bool) (a > b+3);
1340.   sfcpp_expr_3 = (bool) (fun(ob->sys.dat[i++]));
1341.}
```

Nótese que a pesares de que a subexpresión numérica substituída coa variábel auxiliar *sfcpp_expr_2* non é utilizada na expresión orixinal simplificada (liña 1332) ao ser eliminada durante a simplificación da expresión, si é declarada (liña 1329) e o seu valor actualizado (liña 1339) introducindo no código cálculos innecesarios. Este é un aspecto a corrixir en futuras versións do compilador.

```
1342./* Código orixinal (DOWHILE) */
1343.do
1344.{
1345.   <statements>
1346.}
1347.while (↑(500+class::attrib>((↑(c>d*3|| ↓(c||d/3>b)))?12:(26*b))|| ↓c));
1348.
1349./* Código modificado */
1350.// declaración variábeis auxiliares
1351.bool sfcpp_expr_1;
1352.bool sfcpp_expr_2;
1353.bool sfcpp_expr_3;
1354.bool sfcpp_expr_4;
1355.bool sfcpp_expr_5;
1356.do
1357.{
1358.   <statements>
1359.   // actualización variábeis auxiliares
1360.   sfcpp_expr_5 = (bool) (d/3 > b);
1361.   sfcpp_expr_4 = (bool) ((↓c && !sfcpp_expr_5) || (↓sfcpp_expr_5 && !c));
1362.   sfcpp_expr_3 = (bool) (c > d*3);
1363.   sfcpp_expr_2 = (bool) ((!sfcpp_expr_4 && ↑sfcpp_expr_3) ||
1364.                          (↑sfcpp_expr_4 && !sfcpp_expr_3));
1365.   sfcpp_expr_1 = (bool) (500+class::attrib>((sfcpp_expr_2)?12:(26*b)));
1366.}
1367.while (↑sfcpp_expr_1 || (↓c && !sfcpp_expr_1));
```

Como se explica en (§6.3.3.1), cando no resultado da simplificación da expresión con eventos complexa aparece algún evento simple que afecte a algunha das variábeis auxiliares utilizadas para substituír as subexpresións numéricas e con eventos aniñadas, é preciso inserir código adicional que permita detectar os flancos no valor das variábeis auxiliares. Este sería o

caso da variábel *sfcpp_expr_3* (liña 1330), no primeiro dos exemplos anteriores, e das variábeis *sfcpp_expr_1* (liña 1351), *sfcpp_expr_3* (liña 1353), *sfcpp_expr_4* (liña 1354) e *sfcpp_expr_5* (liña 1355), no segundo.

A detección destes eventos realízase na fase *VarInfoHarvester*, e a inserción do código adicional, dacordo á técnica explicada en (§6.3.3), na fase *EventVarTranslator*. O resultado destas modificacións nos exemplos anteriores sería o seguinte:

```
1368./* Código final (FOR2) */
1369.// Variábeis para o cálculo de eventos na subexpresión 3
1370.static bool sfcpp_expr_3_last = false;
1371.bool sfcpp_expr_3_up, sfcpp_expr_3_down;
1372.
1373.// Declaración + iniciación variábeis auxiliares
1374.bool sfcpp_expr_1 = (bool) (3+c > 5);
1375.bool sfcpp_expr_2 = (bool) (a > b+3);
1376.
1377.// Cálculo da subexpresión 3 e dos cambios no seu valor
1378.bool sfcpp_expr_3 = (bool) (fun(ob->sys.dat[i++]));
1379.if (sfcpp_expr_3 != sfcpp_expr_3_last)
1380.{
1381.  sfcpp_expr_3_up = (sfcpp_expr_3_last) ? false : true;
1382.  sfcpp_expr_3_down = (sfcpp_expr_3_last) ? true : false;
1383.  sfcpp_expr_3_last = sfcpp_expr_3;
1384.}
1385.for(<expression>;
1386.     (↓b && !sfcpp_expr_1 && a && !sfcpp_expr_3) ||
1387.      (sfcpp_expr_3_down && !b && a && !sfcpp_expr_3);
1388.     <expression>)
1389.{
1390.   <statements>
1391.   // actualización variábeis auxiliares
1392.   sfcpp_expr_1 = (bool) (3+c > 5);
1393.   sfcpp_expr_2 = (bool) (a > b+3);
1394.   // Actualización da subexpresión 3 e dos cambios no seu valor
1395.   sfcpp_expr_3 = (bool) (fun(ob->sys.dat[i++]));
1396.   if (sfcpp_expr_3 != sfcpp_expr_3_last)
1397.   {
1398.      sfcpp_expr_3_up = (sfcpp_expr_3_last) ? false : true;
1399.      sfcpp_expr_3_down = (sfcpp_expr_3_last) ? true : false;
1400.      sfcpp_expr_3_last = sfcpp_expr_3;
1401.   }
1402.}

1403./* Código final (DOWHILE) */
1404.// Variábeis para o cálculo de eventos nas subexpresións 1,3,4 e 5
1405.static bool sfcpp_expr_1_last = false;
1406.bool sfcpp_expr_1_up, sfcpp_expr_1_down;
1407.static bool sfcpp_expr_3_last = false;
1408.bool sfcpp_expr_3_up, sfcpp_expr_3_down;
1409.static bool sfcpp_expr_4_last = false;
1410.bool sfcpp_expr_4_up, sfcpp_expr_4_down;
1411.static bool sfcpp_expr_5_last = false;
1412.bool sfcpp_expr_5_up, sfcpp_expr_5_down;
1413.
1414.// Declaración variábeis auxiliares
1415.bool sfcpp_expr_1;
1416.bool sfcpp_expr_2;
1417.bool sfcpp_expr_3;
1418.bool sfcpp_expr_4;
1419.bool sfcpp_expr_5;
1420.do
1421.{
1422.   <statements>
```

```
1423.   // Actualización variábeis auxiliares
1424.   sfcpp_expr_5 = (bool) (d/3 > b);
1425.   if (sfcpp_expr_5 != sfcpp_expr_5_last)
1426.   {
1427.      sfcpp_expr_5_up = (sfcpp_expr_5_last) ? false : true;
1428.      sfcpp_expr_5_down = (sfcpp_expr_5_last) ? true : false;
1429.      sfcpp_expr_5_last = sfcpp_expr_5;
1430.   }
1431.
1432.   sfcpp_expr_4 = (bool) ((↓c && !sfcpp_expr_5) || (sfcpp_expr_5_down && !c));
1433.   if (sfcpp_expr_4 != sfcpp_expr_4_last)
1434.   {
1435.      sfcpp_expr_4_up = (sfcpp_expr_4_last) ? false : true;
1436.      sfcpp_expr_4_down = (sfcpp_expr_4_last) ? true : false;
1437.      sfcpp_expr_4_last = sfcpp_expr_4;
1438.   }
1439.
1440.   sfcpp_expr_3 = (bool) (c > d*3);
1441.   if (sfcpp_expr_3 != sfcpp_expr_3_last)
1442.   {
1443.      sfcpp_expr_3_up = (sfcpp_expr_3_last) ? false : true;
1444.      sfcpp_expr_3_down = (sfcpp_expr_3_last) ? true : false;
1445.      sfcpp_expr_3_last = sfcpp_expr_3;
1446.   }
1447.
1448.   sfcpp_expr_2 = (bool) ((!sfcpp_expr_4 && sfcpp_expr_3_up) ||
1449.                          (sfcpp_expr_4_up && !sfcpp_expr_3));
1450.
1451.   sfcpp_expr_1 = (bool) (500+class::attrib>((sfcpp_expr_2)?12:(26*b)));
1452.   if (sfcpp_expr_1 != sfcpp_expr_1_last)
1453.   {
1454.      sfcpp_expr_1_up = (sfcpp_expr_1_last) ? false : true;
1455.      sfcpp_expr_1_down = (sfcpp_expr_1_last) ? true : false;
1456.      sfcpp_expr_1_last = sfcpp_expr_1;
1457.   }
1458.}
1459.while (sfcpp_expr_1_up || (↓c && !sfcpp_expr_1));
```

Recopilación da información das variábeis utilizadas no código ("VarInfoHarvester")

Esta é a fase encargada de detectar as variábeis e eventos simples utilizados no código e recopilar a información que permita en fases posteriores substituílos dacordo ás técnicas explicadas en (§6.3.3). Os tipos de variábeis identificados por esta fase son:

- Variábeis do proceso.
- Variábeis do modelo.
- Estados de activación das etapas do modelo, de sintaxe *X_id*.
- Estados de activación das accións do modelo.
- Variábeis auxiliares para a substitución de temporizadores, de sintaxe *sfcpp_timer_id*. Estas variábeis son introducidas no código como resultado das substitucións realizadas pola fase *TimerPreprocessor* (§6.5.1.5.2).
- Variábeis auxiliares para a substitución de expresións numéricas, de sintaxe *sfcpp_expr_id*. Estas variábeis son introducidas no código como resultado das substitucións realizadas pola fase *EventExpressionCompiler*.

Para cada unha das variábeis identificadas obtense a información seguinte, que será utilizada para determinar o tipo de substitución a realizar:

- Identificador numérico único asignado polo compilador.

- Posición da variábel no código.
- Nome da variábel.
- Tipo de datos da variábel.
- Tipo de variábel (un dos indicados anteriormente).
- Indicador booleano de se a variábel está precedida dun operador de evento.
- Indicador booleano do tipo de operador do que se trata (↑ ou ↓).
- Indicador booleano de se o valor da variábel pode ser consultado.
- Indicador booleano de se o valor da variábel pode ser modificado.

Ao tempo que obtén a información das variábeis, esta fase tamén detecta os seguintes erros:

- Utilización dos operadores de evento con variábeis non booleanas.
- Utilización dos operadores de evento con variábeis booleanas cuxo valor non poda ser consultado, p.e. as saídas do proceso.
- Utilización de variábeis cuxo valor non poda ser consultado cando o código compilado corresponda a unha condición de transición, á condición dunha asociación de acción ou á condición dun temporizador.

Substitución de variábeis e eventos simples ("EventVarTranslator")

Esta é a fase que, utilizando a información recopilada na fase previa, substitúe variábeis e eventos simples polo código que permite acceder aos seus valores en tempo de execución dacordo ás técnicas descritas en (§6.3.3). Esta fase leva rexistro das substitucións xa realizadas de xeito que a mesma variábel auxiliar é utilizada para substituír todas as aparicións no código dunha mesma variábel ou evento simple. Isto permite reducir o número de variábeis auxiliares utilizadas e o numero de accesos á información almacenada pola máquina virtual durante a execución dos modelos.

6.5.1.6. Procesamento das condicións das asociacións ("SFCActionConditionCompiler")

Esta fase comparte subfases coa fase *SFCActionCodeCompiler* (§6.5.1.5), polo que realiza as mesmas operacións, executando as subfases unha vez por cada condición de asociación do modelo.

6.5.1.7. Procesamento das condicións de transición ("SFCReceptivityCompiler")

Esta fase comparte subfases coa fase *SFCActionCodeCompiler* (§6.5.1.5), polo que realiza as mesmas operacións, executando as subfases unha vez por cada condición de transición do modelo.

6.5.1.8. Procesamento das condicións dos temporizadores ("SFCTimerConditionCompiler")

Esta fase comparte subfases coa fase *SFCActionCodeCompiler* (§6.5.1.5), polo que realiza as mesmas operacións, executando as subfases unha vez por cada condición de temporización do modelo.

6.5.1.9. Xeración do código fonte da DLL ("SFCDLLGenerator")

Nesta fase xérase o código fonte da DLL que vai permitir cargar na máquina virtual a información precisa para a execución do modelo. Este código esta formado por dous arquivos

(Figura 6.1), un que contén as declaracións das funcións da DLL (de extensión *.h*) e outro as implementacións das funcións (de extensión *.cpp*). O código C++ xerado para o arquivo que contén as declaracións da DLL é o seguinte:

```
1460. // definicións auxiliares
1461. #define DLL_API __declspec(dllexport)
1462. // interface da DLL
1463. extern "C"
1464. {
1465.    // interface pública común
1466.    DLL_API bool onLoad(void);
1467.    DLL_API void onUnload(void);
1468.    DLL_API deque<module_pointer>& getModules(void);
1469.    // para cada acción (action_name = identificador da acción)
1470.    DLL_API void action_name(RunningPolicy& env);
1471.    // para cada condición de asociación (nnn = identificador da asociación)
1472.    DLL_API bool action_name_nnn(RunningPolicy& env);
1473.    // para cada condición de transición (nnn = identificador da transición)
1474.    DLL_API bool cond_nnn(RunningPolicy& env);
1475.    // para cada temporizador (nnn = identificador do temporizador)
1476.    DLL_API bool timer_nnn(RunningPolicy& env);
1477. }
```

As funcións *onLoad* (liña 1466), *onUnload* (liña 1467) e *getModules* (liña 1468) declaran a interface común a todas as DLLs utilizadas coa máquina virtual (§7.1.1.5), e son utilizadas para a carga e descarga dinámica de módulos executábeis. Ademais inclúense as seguintes declaracións:

1. Unha función por cada acción do modelo (liña 1470). O identificador de cada unha destas funcións é igual ao nome da acción no modelo.
2. Unha función por cada condición de asociación de acción do modelo (liña 1472). O identificador de cada función ten a sintaxe: *action_name_nnn*, sendo *action_name* o nome da asociación (que pode ser o dunha variábel do modelo, variábel de proceso ou acción) e *nnn* o identificador numérico asignado á asociación polo compilador.
3. Unha función por cada receptividade do modelo (liña 1474). O identificador de cada función ten a sintaxe: *cond_nnn*, sendo *nnn* o identificador numérico asignado á receptividade polo compilador (§6.5.1.2.2).
4. Unha función por cada temporizador utilizado nas receptividades do modelo e máis por cada un dos definidos automaticamente polo compilador (§6.5.1.2.1) para a temporización das accións retardadas e limitadas (liña 1476). O identificador de cada función ten a sintaxe: *timer_nnn*, sendo *nnn* o identificador numérico asignado ao temporizador polo compilador (§6.5.1.2.2).

Estas funcións reciben como único parámetro unha instancia da clase *RunningPolicy* (§8.6) que implementa a interface *IVMachineAccess* (§6.3.2) que proporciona acceso aos valores de variábeis e temporizadores almacenados na máquina virtual durante a execución dos modelos. As funcións que representan condicións lóxicas devolven ademais un valor booleano, que é o resultado da avaliación da condición utilizando os valores que variábeis, eventos e temporizadores teñan no momento da chamada á función.

En canto ao arquivo que contén a implementación da DLL, o código xerado é o seguinte:

```
1478. // arquivos de definicións
1479. #include "gescartmodellib.h" // definicións comúns ao compilador e á VM
1480. #include "DLLname.h"         // definicións da DLL
1481. #include "project.h"         // definicións do proxecto
```

```
1482. // declaracións globais
1483. deque<module_pointer> modules;
1484.
1485. /* INTERFACE COMÚN */
1486. // onLoad
1487. DLL_API bool onLoad(void)
1488. {
1489.  // crear información para a execución do modelo
1490.  RTModel* rtmodel = new RTModel();
1491.  if (!rtmodel) return true;
1492.  GESCA_TRY
1493.  {
1494.   RTStaticModel& rtstaticmodel = rtmodel->getStaticModel();
1495.   rtstaticmodel.setKey("model_id"); // almacenar identificador do modelo
1496.
1497.   // para cada declaración de variábel
1498.   modeldatadecl_pointer var = NULL;
1499.   var = new GescaVarDecl<var_data_type>("var_id",
1500.                  (ElementAccess) var_access,
1501.                  (ElementScope) var_scope);
1502.   if (!var) GESCA_THROW_BAD_ALLOC
1503.   rtstaticmodel.getDecls().insert(var);
1504.
1505.   // para cada grafcet parcial
1506.   RTPGInfo* pg = NULL;
1507.   pg = new RTPGInfo(pg_id, "pg_key");
1508.   if (!pg) GESCA_THROW_BAD_ALLOC
1509.   pg->steps.insert(step_id);   // unha liña para cada etapa
1510.   pg->fosteps.insert(step_id); // unha liña para cada etapa con FOs
1511.   rtstaticmodel.addPG(pg);
1512.
1513.   // para cada nodo
1514.   RTNodeInfo* node = NULL;
1515.   node = new RTNodeInfo(node_id, "node_key", pg_id);
1516.   if (!node) GESCA_THROW_BAD_ALLOC
1517.   node->before.insert(node_id);   // unha liña por cada predecesor
1518.   node->after.insert(node_id);    // unha liña por cada sucesor
1519.   rtstaticmodel.addNode(node);    // addInitialNode se é unha etapa inicial
1520.
1521.   // para cada condición de transición
1522.   RTReceptivityInfo* tinfo = NULL;
1523.   tinfo = new RTReceptivityInfo(rec_id, "cond_rec_id", source_flag);
1524.   if (!tinfo) GESCA_THROW_BAD_ALLOC
1525.   tinfo->vars.insert("var_name"); // unha liña para cada variábel
1526.   rtstaticmodel.addReceptivity(tinfo);
1527.   rtmodel->addCondition("cond_rec_id",
1528.               cond_rec_id);   // referencia ao código
1529.
1530.   // para cada asociación
1531.   RTActionInfo* ainfo = NULL;
1532.   ainfo = new RTActionInfo(assoc_id, step_id, "assoc_name",
1533.               (ActionType) assoc_type, "assoc_indicator",
1534.               assoc_delay, assoc_limit, "assoc_name_nnn",
1535.               (TimeScale) assoc_tscale,
1536.               (RTActionType) assoc_rttype);
1537.   if (!ainfo) GESCA_THROW_BAD_ALLOC
1538.   ainfo->vars.insert("var_name"); // unha liña para cada variábel
1539.   rtstaticmodel.addAction(ainfo);
1540.   rtmodel->addCondition("assoc_name_nnn",
1541.               assoc_name_nnn); // referencia ao código
1542.
1543.   // para cada acción
1544.   var = new GescaVarDecl<bool>("action_name",
1545.                (ElementAccess) 2, // acceso R/W
1546.                (ElementScope) 2); // alcance
```

```
1547.   if (!var) GESCA_THROW_BAD_ALLOC
1548.   rtstaticmodel.getDecls().insert(var);
1549.   rtmodel->addAction("action_name", action_name); // referencia ao código
1550.
1551.   // para cada orde de forzado
1552.   RTFOInfo* foinfo = NULL;
1553.   foinfo = new RTFOInfo(pg_id, step_id, forcedpg_id,
1554.                (FOrderType) fo_type);
1555.   if (!foinfo) GESCA_THROW_BAD_ALLOC
1556.   foinfo->steps.insert(step_id); // unha liña por cada etapa forzada
1557.   rtstaticmodel.addFO(foinfo);
1558.
1559.   // xerarquía de forzado
1560.   RTSituation level;
1561.   rtstaticmodel.initFOLevels(num_folevels);
1562.   // para cada nivel da xerarquía
1563.   level.insert(step_id); // unha liña por cada etapa no nivel
1564.   rtstaticmodel.addFOLevel(num_level, level);
1565.   level.clear();
1566.
1567.   // para cada temporizador
1568.   RTTimerInfo* timer = NULL;
1569.   var = new GescaVarDecl<bool>("sfcpp_timer_nnn",
1570.                (ElementAccess) 2, // acceso R/W
1571.                (ElementScope) 2); // alcance
1572.   if (!var) GESCA_THROW_BAD_ALLOC
1573.   rtstaticmodel.getDecls().insert(var);
1574.   timer = new RTTimerInfo(timer_id, "timer_nnn", timer_t1, timer_t2);
1575.   if (!timer) GESCA_THROW_BAD_ALLOC
1576.   timer->vars.insert(var_name); // unha liña para cada variábel
1577.   rtstaticmodel.addTimer(timer);
1578.   rtmodel->addCondition("timer_nnn", timer_nnn); // referencia ao código
1579.
1580.   // almacenar o modelo no conxunto de módulos
1581.   modules.push_back(rtmodel);
1582.  }
1583.  GESCA_CATCH_ALL
1584.  {
1585.   delete rtmodel;
1586.   return true;
1587.  }
1588.  return false;
1589. }
1590.
1591. // onUnload
1592. DLL_API void onUnload(void)
1593. {
1594.   for (deque<module_pointer>::iterator module = modules.begin();
1595.        module != modules.end();
1596.        ++module)
1597.     delete *module;
1598.   modules.clear();
1599. }
1600.
1601. // getModules
1602. DLL_API deque<module_pointer>& getModules(void)
1603. {
1604.  return modules;
1605. }
```

```
1606. /* CÓDIGO DO MODELO */
1607. // para cada acción
1608. DLL_API void action_name(RunningPolicy& env)
1609. {
1610.    // código da acción
1611. }
1612.
1613. // para cada condición de asociación
1614. DLL_API bool action_name_nnn(RunningPolicy& env)
1615. {
1616.    // código da condición de asociación
1617. }
1618.
1619. // para cada condición de transición
1620. DLL_API bool cond_nnn(RunningPolicy& env)
1621. {
1622.    // código da condición de transición
1623. }
1624.
1625. // para cada temporizador
1626. DLL_API bool timer_nnn(RunningPolicy& env)
1627. {
1628.    // código da condición do temporizador
1629. }
```

Nas liñas 1479-1481 inclúense os arquivos que conteñen as declaracións do ambiente de execución da máquina virtual, as da DLL e as de usuario, respectivamente. A implementación das funcións que permiten a carga e descarga dinámica de módulos executábeis na máquina virtual está nas liñas 1487-1605. A función *onLoad* (liña 1487), chamada pola máquina virtual durante a carga da DLL, inicia a información do modelo e insírea na cola de módulos —variábel global *modules* declarada na liña 1483—. A máquina virtual accede aos módulos almacenados nesta variábel mediante a función *getModules* (liña 1602). Os módulos son eliminados na función *onUnload* (liña 1592), chamada pola máquina virtual cando a DLL é descargada. Ademais inclúese a implementación das funcións que conteñen o código de accións (liña 1608), condicións de asociación (liña 1614), condicións de transición (liña 1620) e condicións de temporización (liña 1626) obtido como resultado das substitucións explicadas en (§6.3.3).

A implementación da función *onLoad*, na que se inicia a información do modelo utilizando o formato explicado en (§6.4), contén o código seguinte:

1. Na liña 1490 créase unha instancia da clase *RTModel* (Figura 6.8), que vai conter a información para a execución do modelo.
2. Na liña 1494 obtense unha referencia á información estática do modelo.
3. Na liña 1495 asígnase un identificador ao modelo. Este identificador é utilizado na máquina virtual para identificar o modelo no caso de ter varios cargados simultaneamente na memoria.
4. Nas liñas 1498-1503 insírese, para cada variábel do modelo, o código para crear a súa declaración utilizando a información recopilada polo compilador: nome, tipo, acceso e alcance da variábel; e para inserila no modelo.
5. Nas liñas 1506-1511 insírese, para cada grafcet parcial, o código para crear unha instancia da clase *RTPGInfo* (Figura 6.8) utilizando a información recopilada polo compilador: identificador numérico e alfanumérico do grafcet parcial; para engadirlle a información das etapas contidas no grafcet parcial e das que teñen ordes de forzado asociadas (unha liña de código por etapa); e para inserir a información no modelo.

6. Nas liñas 1514-1519 insírese, para cada nodo do modelo, o código para crear unha instancia da clase *RTNodeInfo* (Figura 6.8) utilizando a información recopilada polo compilador: identificador numérico e alfanumérico da etapa, e identificador numérico do grafcet parcial que contén a etapa; para engadirlle a información dos nodos que o suceden e que o preceden na secuencia de control (unha liña de código por cada un); e para inserir a información no modelo (as etapas iniciais son inseridas no modelo utilizando un método diferente —*addInitialNode*— ao utilizado para os demais nodos —*addNode*—).
7. Nas liñas 1522-1528 insírese, para cada receptividade do modelo, o código para crear unha instancia da clase *RTReceptivityInfo* (Figura 6.8) utilizando a información recopilada polo compilador: identificador numérico da receptividade, nome da función que contén o código da receptividade e indicador booleano de se a receptividade está asociada a unha transición fonte; para engadirlle a información das variábeis utilizadas no código da receptividade (unha liña de código por cada unha); e para inserir a información no modelo (insírese a instancia da clase *RTReceptivityInfo* e a información que permite obter un apuntador á función que contén o código da receptividade dado o seu nome).
8. Nas liñas 1531-1541 insírese, para cada asociación de acción do modelo, o código para crear unha instancia da clase *RTActionInfo* (Figura 6.8) utilizando a información recopilada polo compilador: identificador numérico da asociación, identificador numérico da etapa á que está asociada, nome da variábel booleana modificada pola asociación, cualificador da asociación, nome da variábel utilizada como indicador, valores de retardo e límite en asociacións temporizadas, nome da función que contén o código da condición da asociación, indicador de se a asociación pode ser aplicada en situacións inestábeis e valor do tipo interno de asociación asignado polo compilador; para engadirlle a información das variábeis utilizadas no código da condición da asociación (unha liña de código por cada unha); e para inserir a información no modelo (insírese a instancia da clase *RTActionInfo* e a información que permite obter un apuntador á función que contén o código da condición da asociación dado o seu nome).
9. Nas liñas 1544-1549 insírese, para cada acción do modelo, o código para crear unha variábel booleana co nome da acción que almacenará o seu valor de activación e para inserir a información no modelo (insírese a declaración da variábel e a información que permite obter un apuntador á función que contén o código da acción dado o seu nome).
10. Nas liñas 1552-1557 insírese, para cada orde de forzado do modelo, o código para crear unha instancia da clase *RTFOInfo* (Figura 6.8) utilizando a información recopilada polo compilador: identificador numérico do grafcet parcial que contén a etapa á que está asociada a orde de forzado, identificador numérico desta etapa, identificador numérico do grafcet parcial forzado e identificadores numéricos das etapas forzadas (unha liña de código por cada etapa forzada); e para inserir a información no modelo.
11. Nas liñas 1560-1565 insírese o código para iniciar o número de niveis da xerarquía de forzado do modelo. Ademais, para cada nivel, insírese o código para engadir o identificador numérico das etapas pertencentes ao nivel que teñen ordes de forzado asociadas (unha liña de código por cada etapa) e para inserir a información no modelo.
12. Nas liñas 1568-1578 insírese, para cada temporizador do modelo, o código para crear unha variábel booleana de nome *sfcpp_timer_nnn* (*nnn* = identificador numérico do temporizador) que almacenará o seu valor, para crear unha instancia da clase *RTTimerInfo* (Figura 6.8) utilizando a información recopilada polo compilador: identificador numérico do temporizador, nome da función que contén o código da condición do temporizador, valores de retardo e límite; para engadirlle a información das variábeis utilizadas no código

da condición do temporizador (unha liña de código por cada unha) e para inserir a información no modelo (insírese a instancia da clase *RTTimerInfo*, a declaración da variábel e a información que permite obter un apuntador á función que contén o código da condición do temporizador dado o seu nome).

13. Na liña 1581 insírese a información do modelo na cola de módulos.

No caso de que algunha das operacións anteriores provoque un erro durante a execución, lanzase unha excepción que é capturada nas liñas 1583-1587, onde se libera a memoria utilizada ate o momento e sáese da función *onLoad* devolvendo un valor de erro (*true*).

6.5.1.10. Compilación da DLL ("SFCCPPCompiler")

Nesta fase realízase a compilación e enlazado do código fonte da DLL xerado nas fases previas. Esta fase é unha clase "wrapper" que capsula o acceso a un compilador externo utilizando a técnica explicada en (§6.1.3). Esta clase presupón a existencia dun arquivo denominado *compile.bat* que estará almacenado no subdirectorio do directorio *compilers* indicado nas opcións de configuración (§6.1.1).

6.6. Conclusións

Neste capítulo tratáronse os aspectos relacionados coa compilación dos modelos Grafcet para obter unha DLL que permita cargalos dinamicamente e executalos na máquina virtual que implementa o intérprete de modelos Grafcet. Describiuse o formato utilizado para representar os modelos de forma optimizada para a súa execución. Analizáronse os aspectos a considerar cando se utiliza unha linguaxe de alto nivel como o C++, estendida cos operadores Grafcet de evento e temporización, na especificación das accións e receptividades do modelo. Fíxose fincapé nas modificacións realizadas na gramática da linguaxe para incluír os novos operadores e nas substitucións que é preciso realizar no código de accións e receptividades para poder compilalas cun compilador C++ externo.

No referente á implementación do compilador Grafcet, describiuse a súa arquitectura lóxica, composta por unha estructura de fases que acceden durante a compilación á información interna almacenada nun único punto accesíbel a todas as fases. Esta arquitectura permite a modificación da estructura de fases ou a reimplementación dunha fase concreta sen que iso afecte ao resto nin a estructura do compilador. Nesta primeira versión do compilador a implementación das fases presupón que a linguaxe de programación utilizada é o C++, sen embargo partindo da estructura deseñada, é fácil en futuras versións engadir novas implementacións das fases que permitan utilizar linguaxes de alto nivel diferentes. Tamén se describiu a técnica que permite utilizar aplicacións externas, como un compilador C++ por exemplo, sen que sexa necesario modificar o compilador Grafcet se se cambia de aplicación.

En canto ás melloras a realizar, resúmense aquí algunhas das xa comentadas nos contidos do capítulo:

1. Eliminar a restricción de que as expresións con eventos complexas teñan que ir entre parénteses (§6.3.1.1).
2. Reducir as restriccións impostas nas expresións que poden utilizarse cos operadores de evento e temporización (§6.3.1.2).
3. Permitir a utilización de temporizadores no código das accións (§6.5.1.5.2).
4. Optimizar a substitución das subexpresións numéricas ao simplificar expresións con eventos complexas (§6.3.3.1). Na implementación actual non se trata o caso no que como

resultado da simplificación se obtén unha expresión na que non se utiliza a variábel auxiliar que substitúe á subexpresión numérica. Tampouco se ten en conta se a subexpresión é constante cando se obtén na expresión simplificada un evento que afecte á variábel auxiliar.

5. Eliminar a restricción de que o valor das saídas do proceso unicamente poda ser modificado mediante o operador de asignación simple (§6.3.3.3).

Outras optimizacións, que reducirían o tempo necesario para compilar un modelo, son as seguintes:

1. Substituír a fase que chama á aplicación externa Sidoni por outra que implemente directamente a simplificación de expresións booleanas. Isto eliminaría o tempo necesario para executar a aplicación externa cada vez que hai que simplificar unha expresión.
2. Deseñar unha técnica que permita preprocesar todo o código do modelo nunha única chamada ao preprocesador. Na implementación actual o preprocesador é executado unha vez por cada acción, condición de transición, condición de asociación e condición de temporización do modelo, co consumo de tempo que isto implica.

Capítulo 7. A máquina virtual

A máquina virtual é un ambiente "software" que proporciona soporte á execución de aplicacións que interaccionen cun proceso físico mediante o intercambio de eventos discretos a través dun ou máis dispositivos de E/S. Dende o punto de vista do usuario, a máquina virtual proporciona un conxunto de servicios aos que se accede a través dun enlace de comunicacións que permiten o control remoto do seu funcionamento e a modificación da súa configuración. Dende o punto de vista das aplicacións, a máquina virtual proporciona unha interface común de acceso ás funcionalidades do sistema no que se executan as aplicacións, como por exemplo: a notificación de eventos, a xestión das temporizacións ou o acceso aos valores das variábeis do proceso.

Aínda que inicialmente foi deseñada co obxectivo de proporcionar soporte a un intérprete no que executar os modelos Grafcet, pode ser utilizada igualmente para a implementación de intérpretes doutros formalismos para a especificación de DEDS, como: RdPI, StateCharts, etc. ou doutras aplicacións que interaccionen cun proceso físico como, por exemplo, un emulador PLC. Como parte do desenvolvemento da arquitectura da máquina virtual implementouse unha libraría que inclúe diferentes técnicas para facilitar a creación de aplicacións concorrentes, portábeis e reconfigurábeis dinamicamente.

O resto deste capítulo está organizado da forma seguinte: no apartado (§7.1) descríbense a arquitectura da máquina virtual e os mecanismos utilizados para dar soporte á súa estructura flexíbel e concorrente; en (§7.2) explícase o subsistema que xestiona a interacción cos dispositivos físicos e permite simular as magnitudes dun proceso; no apartado (§7.3) descríbese o núcleo da máquina virtual no que se almacenan os valores do proceso e xestiónanse temporizadores e eventos; os servicios proporcionados pola máquina virtual aos clientes remotos para a xestión da súa configuración e o control do seu funcionamento son explicados no apartado (§7.4); no apartado (§7.5) detállase a interacción entre as aplicacións executadas pola máquina virtual e o núcleo desta; e finalmente, no apartado (§7.6) recóllense as conclusións do capítulo.

7.1. A arquitectura da máquina virtual

O deseño da arquitectura da máquina virtual ten como obxectivo proporcionar os servicios precisos para a execución de aplicacións baseadas no intercambio de eventos discretos cun proceso físico. Os requisitos considerados no deseño foron os seguintes:

1. A arquitectura debía ser flexíbel, de xeito que fose posíbel utilizar diferentes dispositivos de E/S, configurar a interacción entre a máquina virtual e o proceso, e executar distintos tipos de aplicacións.

2. A implementación debía ser facilmente portábel, reducindo no posíbel as dependencias de aspectos específicos dos sistemas utilizados, facilitando así a aplicación da máquina virtual en ambientes de control heteroxéneos.

Para cumprir estes requisitos utilizáronse dúas solucións complementarias:

1. A arquitectura está baseada na utilización de módulos (§7.1.1). Os módulos son elementos "software" que poden ser cargados e descargados dinamicamente na memoria, substituídos en tempo de execución, utilizados como base para formar estructuras complexas mediante agregación e composición, etc.
2. Definiuse unha capa software (Figura 7.1) que proporciona unha interface abstracta das funcionalidades proporcionadas polo sistema operativo, dispositivos físicos e redes de comunicación nos que se execute a máquina virtual. A implementación desta capa basease na utilización de patróns de deseño [67] e na definición de interfaces mediante clases abstractas e métodos virtuais, de xeito que a portabilidade da máquina virtual conséguese implementando as interfaces definidas nos diferentes sistemas a utilizar. As funcionalidades proporcionadas pola capa de abstracción do sistema comprenden: procesos e mecanismos de comunicación entre procesos (§7.1.2); "drivers" de E/S (§7.2.1); temporizadores (§7.3.2); e interfaces de comunicación (§7.4.3).

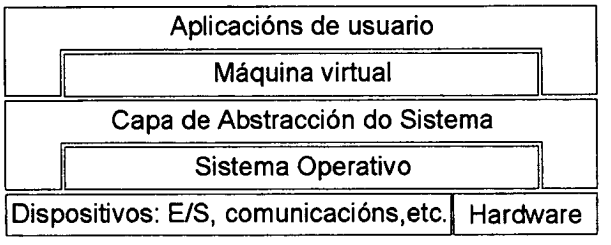


Figura 7.1. Estructura de capas da máquina virtual.

O resto deste apartado estructúrase do xeito seguinte: os detalles dos dous mecanismos básicos (módulos e procesos) utilizados no deseño e implementación da máquina virtual explícanse en (§7.1.1) e (§7.1.2), respectivamente; a arquitectura da máquina virtual descríbese en (§7.1.3); e a súa implementación en (§7.1.4).

7.1.1. Estructuración da arquitectura: os módulos

O elemento básico da arquitectura da máquina virtual é o módulo. Os módulos son elementos "software" que poden ser creados ou cargados dinamicamente na memoria dende DLLs en tempo de execución. O seu uso facilita a implementación de estructuras complexas e de comportamentos flexíbeis mediante a agregación e a substitución dinámica de módulos sen que sexa preciso deter a execución da aplicación. A Figura 7.2 mostra o diagrama das interfaces e clases utilizadas para declarar e implementar as funcionalidades dispoñíbeis na creación e xestión de arquitecturas software baseadas en módulos. No resto deste apartado descríbense estas clases e as funcionalidades que proporcionan e danse exemplos da súa utilización.

7.1.1.1. Módulos simples: a interface *IModule*

A clase abstracta *IModule* declara a interface básica dun módulo. Esta interface está formada unicamente por dous métodos que proporcionan acceso aos identificadores alfanuméricos do

módulo, un que indica o seu tipo e outro que identifica univocamente ao módulo entre os do seu mesmo tipo. O código da declaración é o seguinte:

```
1630. // Interface dun módulo
1631. struct IModule : public ptrSeqElem
1632. {
1633.  virtual string getKey() const = 0;
1634.  virtual string getType() const = 0;
1635. };
1636.
1637. // Declaracións auxiliares
1638. typedef IModule module;
1639. typedef IModule* module_pointer;
1640. typedef id2ptrSeq<module> module_map;
1641. typedef id2ptrSeq<module>::iterator module_iterator;
```

Nótese que os módulos son derivados da clase *ptrSeqElem*, de xeito que poidan ser almacenados en memoria utilizando a colección *id2ptrSeq* (§B.1).

7.1.1.2. Módulos complexos: a interface IStructuredModule

A clase abstracta *IStructuredModule* declara a interface a implementar polos módulos complexos cuxa estructura se forma mediante a agregación doutros módulos que poden ser substituídos dinamicamente durante a execución da aplicación. O código da declaración desta interface é o seguinte:

```
1642. struct IStructuredModule
1643. {
1644.  // consulta información da estructura actual
1645.  virtual void getStructureInfo(deque<VMModuleInfo>& structure) const = 0;
1646.  // almacén de módulos
1647.  virtual IModuleStore* getModuleStore() const = 0;
1648.  virtual void setModuleStore(IModuleStore* store) = 0;
1649.  // almacén de configuracións
1650.  virtual IConfigurationStore* getConfigurationStore() const = 0;
1651.  virtual void setConfigurationStore(IConfigurationStore* store) = 0;
1652.  // activar/desactivar configuración
1653.  virtual bool activateCurrentConf() = 0;
1654.  virtual bool deactivateCurrentConf() = 0;
1655. };
```

A interface declara métodos para consultar a estructura actual do módulo, asignar e obter apuntadores tanto ao almacén de módulos (§7.1.1.3) como ao de configuracións (§7.1.1.4) utilizados polo módulo, e activar e desactivar a configuración actual do módulo. A clase *StructuredModule* (Figura 7.2) proporciona a implementación por defecto dos métodos *getModuleStore* (liña 1647), *setModuleStore* (liña 1648), *getConfigurationStore* (liña 1650), e *setConfigurationStore* (liña 1651). Os outros métodos serán redefinidos nas clases derivadas desta interface.

A información dos tipos e nomes dos módulos que forman a estructura dun módulo complexo denomínase *configuración*. Cada módulo complexo pode adoptar diferentes configuracións, aínda que en cada instante só unha estea activa. As configuracións son representadas mediante a clase *ModuleConfiguration* (Figura 7.2) e conteñen a información necesaria para que os módulos complexos podan organizar dinamicamente a súa estructura interna. Cada configuración contén, por cada tipo de módulo utilizado, unha instancia da clase *ConfigurationItem* na que se almacena o nome do tipo de módulo, a cantidade de módulos dese

tipo utilizados[69] e, no caso de especificarse unha cantidade fixa, os nomes dos módulos utilizados. Os detalles dos cambios de configuración son implementados nos métodos *activateCurrentConf* (liña 1653) e *deactivateCurrentConf* (liña 1654) nas clases derivadas.

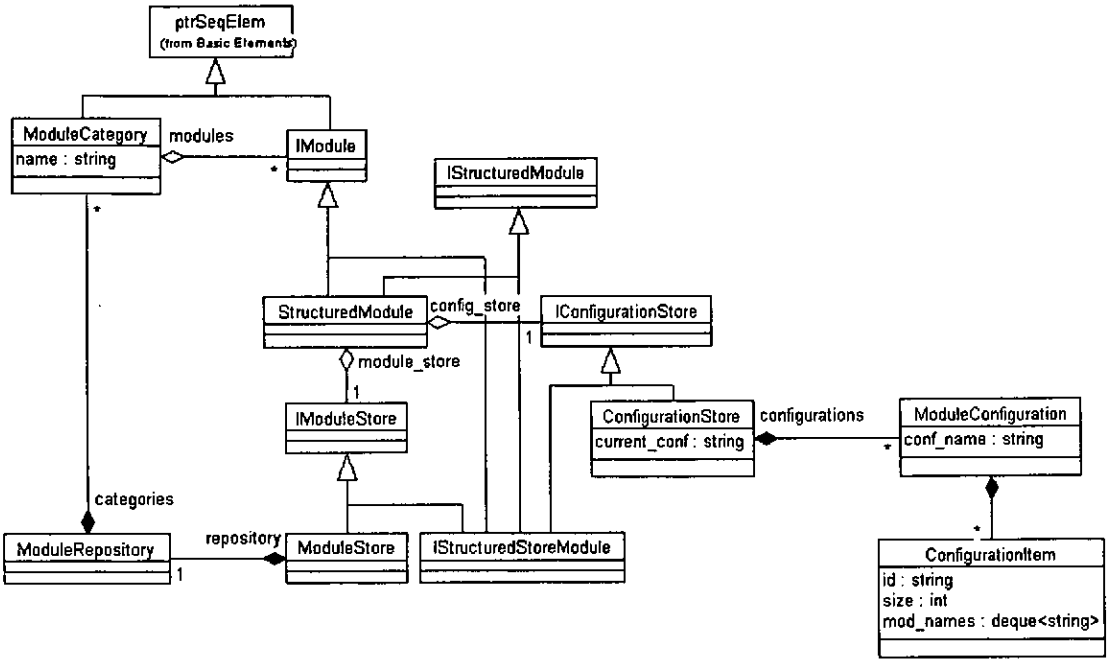


Figura 7.2. Diagrama das clases que definen os módulos e as súas funcionalidades.

A consulta sobre a estructura actual dun módulo complexo realízase mediante o método *getStructureInfo* (liña 1645). A implementación deste método é recursiva e devolve para cada módulo complexo agregado un rexistro contendo o nome e tipo do módulo, o nome do almacén de módulos que utiliza (§7.1.1.3) e os nomes dos módulos agregados que compoñen a súa estructura. O código seguinte mostra a declaración destes rexistros:

```
1656.struct VMModuleInfo
1657.{
1658.  string name;           // nome do módulo complexo
1659.  string type;           // tipo
1660.  string store;          // nome do almacén de módulos que utiliza
1661.  deque<string> parts;   // módulos agregados que forman a súa estructura
1662.};
```

7.1.1.3. Almacéns de módulos: a interface *IModuleStore*

A clase abstracta *IModuleStore* declara a interface para a xestión do almacenamento de múltiples módulos en memoria organizados formando categorías en función do seu tipo. O código da súa declaración é o seguinte:

```
1663.// Interface dun almacén de módulos
1664.struct IModuleStore
1665.{
1666.   // categorías
1667.   virtual void initCategories() = 0;
1668.   virtual bool isCategory(const string& name) const = 0;
```

[69] Na implementación actual aceptanse duas posibilidades, ou unha cantidade fixa ou unha cantidade non especificada (0..N módulos).

```
1669.    virtual void getCategoryNames(deque<string>& names) const = 0;
1670.    virtual void addCategory(const string& name) = 0;
1671.    virtual void removeCategory(const string& name) = 0;
1672.    virtual void clearCategory(const string& name) = 0;
1673.    // módulos
1674.    virtual bool isModule(const string& type,
1675.                          const string& name) const = 0;
1676.    virtual bool getModuleNames(const string& type,
1677.                                deque<string>& names) const = 0;
1678.    virtual module_pointer getModule(const string& type,
1679.                                     const string& name) = 0;
1680.    virtual module_iterator addModule(module_pointer module) = 0;
1681.    virtual bool removeModule(const string& type, const string& name) = 0;
1682.    virtual void clearAllModules() = 0;
1683. };
```

Como pode verse a interface declara métodos para crear, consultar e eliminar categorías (liñas 1667-1672) e para engadir, eliminar e recuperar os módulos almacenados nas categorías existentes (liñas 1674-1682). A clase *ModuleStore* (Figura 7.2) proporciona a implementación por defecto desta interface utilizando un almacén de módulos, representado pola clase *ModuleRepository*, que pode conter varias categorías, representadas mediante a clase *ModuleCategory*. Cada categoría almacena un conxunto de módulos do mesmo tipo. O nome da categoría coincidirá co do tipo de módulos que almacene.

7.1.1.4. Almacéns de configuracións: a interface *IConfigurationStore*

A clase abstracta *IConfigurationStore* declara a interface para a xestión do almacenamento de múltiples configuracións en memoria. O código da súa declaración é o seguinte:

```
1684. struct IConfigurationStore
1685. {
1686.    // iniciación das configuracións
1687.    virtual void initConfigurations() = 0;
1688.    virtual bool setDefaultConf() = 0;
1689.    // engadir, eliminar e modificar configuracións
1690.    virtual bool addConfiguration(const ModuleConfiguration& c) = 0;
1691.    virtual bool removeConfiguration(const string& id) = 0;
1692.    virtual bool updateConfiguration(const string& id,
1693.                                     const ModuleConfiguration& c) = 0;
1694.    // consulta de configuracións
1695.    virtual bool isValidConfiguration(const ModuleConfiguration& c) const = 0;
1696.    virtual const ModuleConfiguration&
1697.         getConfiguration(const string& name) const = 0;
1698.    virtual void getConfigurationNames(deque<string>& names) const = 0;
1699.    // configuración actual
1700.    virtual const string& getCurrentConf() const = 0;
1701.    virtual bool setCurrentConf(const string& id = "NONE") = 0;
1702. };
```

Como pode verse esta interface proporciona métodos para a iniciación (liñas 1687-1688), consulta (liñas 1695-1698), inserción (liña 1690) e eliminación (liña 1691) de configuracións, así como para consultar (liña 1700) e seleccionar (liña 1701) a configuración actual. A clase *ConfigurationStore* (Figura 7.2) proporciona a implementación por defecto da maioría dos métodos desta interface —excepto *initConfigurations* (liña 1687), *isValidConfiguration* (liña 1695) e *setDefaultConf* (liña 1688)—.

Ademais das interfaces e clases xa comentados tamén se definiu a interface *IStructuredStoreModule* (Figura 7.2), derivada simultaneamente das catro interfaces explicadas anteriormente: *IModule*, *IStructuredModule*, *IModuleStore* e *IConfigurationStore*, para ser

utilizada como base dos módulos complexos que sexan utilizados simultaneamente como almacéns de módulos e configuracións.

7.1.1.5. Carga e descarga dinámica de módulos

Unha funcionalidade adicional proporcionada polos módulos é a posibilidade de cargalos e descargalos dinamicamente en memoria mediante a utilización de DLLs, o que proporciona un mecanismo simple para a extensión e modificación das características dunha aplicación en tempo de execución. Este mecanismo é utilizado con frecuencia na implementación da máquina virtual, como por exemplo para o manexo dos "drivers" de E/S, dos modelos Grafcet ou das políticas de evolución. O código seguinte mostra a interface pública dunha DLL que proporcione soporte á carga e descarga dinámica de módulos:

```
1703. #define DLL_API __declspec(dllexport)
1704. extern "C"
1705. {
1706.  DLL_API bool onLoad(void);
1707.  DLL_API void onUnload(void);
1708.  DLL_API deque<module_pointer>& getModules(void);
1709. }
```

Esta interface está formada por tres funcións:

1. *onLoad* (liña 1706), función executada no momento de cargar a DLL en memoria na que se inician os módulos que serán posteriormente cargados na aplicación.
2. *onUnload* (liña 1707), función executada no momento de descargar a DLL. Nela impleméntanse as operacións necesarias para liberar a memoria e recursos utilizados polos módulos.
3. *getModules* (liña 1708), función que permite acceder aos módulos contidos na DLL.

O seguinte código mostra unha implementación típica destas funcións:

```
1710. // colección de módulos
1711. deque<module_pointer> modules;
1712.
1713. DLL_API bool onLoad(void)
1714. {
1715.  Module* module_1 = NULL;
1716.  Module* module_2 = NULL;
1717.  try
1718.  {
1719.   // crear módulos
1720.   Module* module_1 = new Module("MODULE_1", "TYPE");
1721.   if (!module_1) throw;
1722.   Module* module_2 = new Module("MODULE_2", "TYPE");
1723.   if (!module_2) throw;
1724.   // almacenar os módulos na colección
1725.   modules.push_back(module_1);
1726.   modules.push_back(module_2);
1727.  }
1728.  catch(...)
1729.  {
1730.   // eliminar módulos en caso de erro
1731.   if (module_1) delete module_1;
1732.   if (module_2) delete module_2;
1733.   return false;
1734.  }
1735.  return true;
1736. }
```

```
1737. DLL_API void onUnload(void)
1738. {
1739.    for (deque<module_pointer>::iterator module = modules.begin();
1740.            module != modules.end();
1741.          ++module)
1742.       delete *module;
1743.    modules.clear();
1744. }
1745.
1746. DLL_API deque<module_pointer>& getModules(void)
1747. {
1748.    return modules;
1749. }
```

A variábel global *modules* (liña 1711) é unha cola na que se almacenarán os módulos contidos na DLL. Na implementación da función *onLoad* (liñas 1713-1736) créanse os módulos e almacénanse na cola *modules* (liñas 1719-1726). En caso de detectarse algún erro lánzase unha excepción que é procesada nas liñas 1730-1733, eliminando a memoria ocupada polos módulos. Na función *onUnload* (liñas 1737-1744), executada cando a DLL é descargada da memoria, elimínanse todos os módulos almacenados en *modules*. Por último, a función *getModules* (liña 1746) devolve unha referencia á colección de módulos da DLL.

O acceso á DLL dende unha aplicación faise mediante unha instancia da clase *DLLInfo*. Esta clase oculta os detalles da carga e descarga de DLLs, ofrecendo unha interface independente do sistema operativo utilizado. A súa declaración é a seguinte:

```
1750. class DLLInfo
1751. {
1752. public:
1753.    // carga/descarga da DLL
1754.    DLLInfo(const string& _name, const string& _path);
1755.    ~DLLInfo();
1756.    // métodos públicos da DLL
1757.    bool load();
1758.    bool unload();
1759.    bool getModules(module_map& _modules);
1760. };
```

O constructor (liña 1754) e o destructor (liña 1755) da clase son os que implementan respectivamente a carga e descarga en memoria da DLL. O resto dos métodos permiten o acceso ás funcións da interface pública da DLL explicadas anteriormente.

7.1.1.6. Exemplo da utilización de módulos

Neste apartado descríbese un exemplo de como son utilizadas as funcionalidades proporcionadas polos módulos para implementar unha aplicación coa arquitectura flexíbel da Figura 7.3.a. A arquitectura da aplicación está formada por dous módulos, un simple, que sirve de almacén dos módulos creados dinamicamente en memoria, e outro complexo cunha estructura formada pola agregación de dous módulos, un complexo de tipo A e outro simple de tipo B. O módulo complexo de tipo A ten á súa vez unha estructura formada pola agregación dun módulo simple de tipo C. A Figura 7.3.b mostra as configuracións válidas que son aceptadas na arquitectura indicada, e o diagrama de obxectos da Figura 7.3.c a súa implementación utilizando as clases explicadas anteriormente. Nótese que o módulo *store* é utilizado como almacén das configuracións válidas do módulo *main* e este, á súa vez, como almacén das configuracións válidas dos módulos tipo A. Os diagramas de obxectos das configuracións válidas da arquitectura son mostrados na Figura 7.3.d.

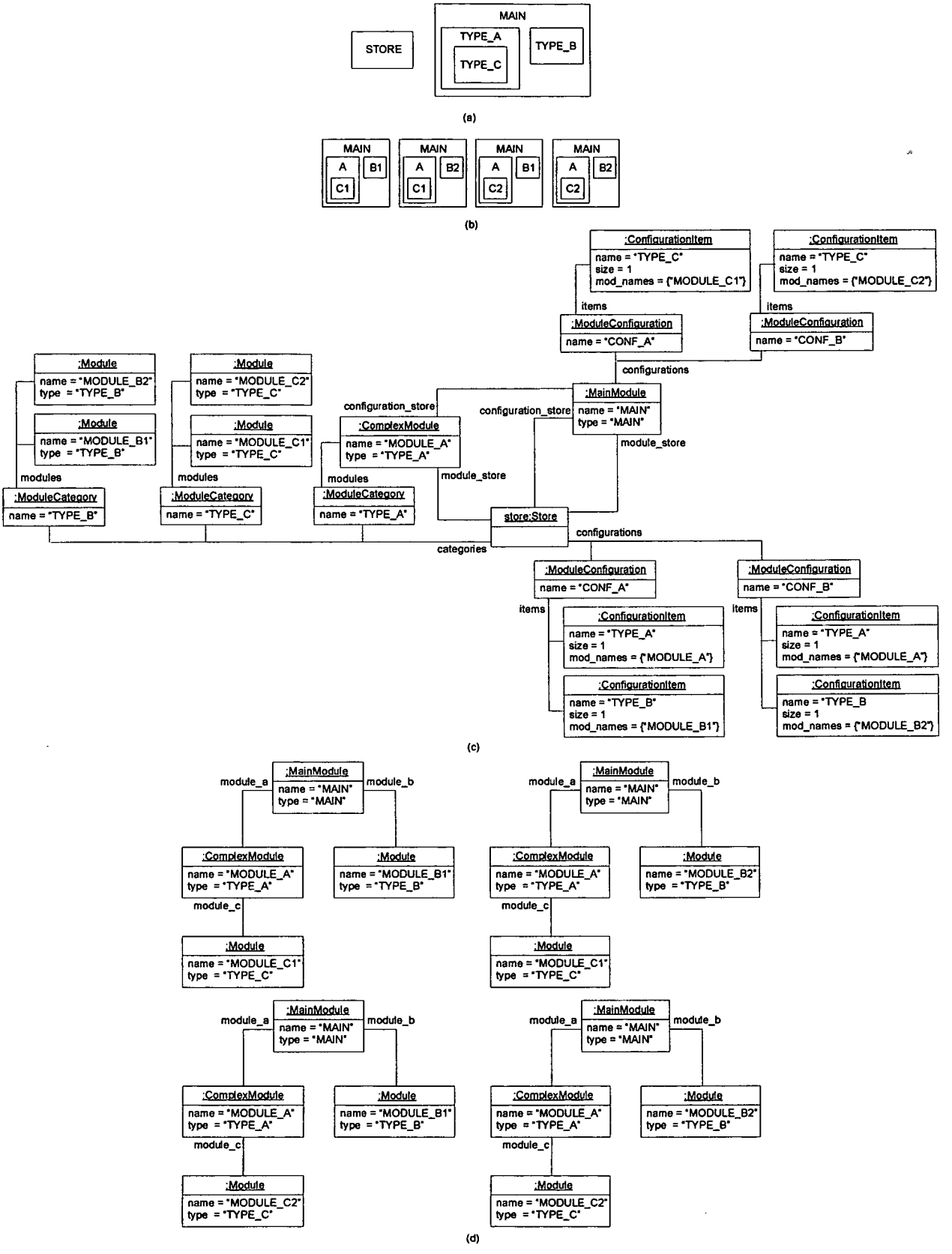


Figura 7.3. Exemplo de utilización dos módulos: a) arquitectura flexíbel da aplicación; b) configuracións válidas; c) diagrama de obxectos da implementación; e d) diagramas de obxectos das configuracións válidas.

O código que implementa a arquitectura do exemplo é o seguinte:

```
1761. // Módulo simple
1762. class Module : public IModule
1763. {
1764.  string name;
1765.  string type;
1766. public:
1767.  Module(const string& n, const string& t):name(n), type(t){}
1768.  string getKey() const { return name; }
1769.  string getType() const { return type; }
1770. };
1771.
1772. // Módulo complexo (tipo A)
1773. class ComplexModule : public StructuredModule, public Module
1774. {
1775.  // módulos agregados
1776.  IModule* module_C;
1777. public:
1778.  ComplexModule(const string& n, IModuleStore* mstore, IConfigurationStore* cstore);
1779.  void getStructureInfo(deque<VMModuleInfo>& s) const;
1780.  bool activateCurrentConf();
1781.  bool deactivateCurrentConf();
1782. };
1783.
1784. ComplexModule::ComplexModule(const string& n, IModuleStore* mstore,
1785.                  IConfigurationStore* cstore)
1786.   : StructuredModule(mstore, cstore), Module(n, "TYPE_A") {}
1787.
1788. void ComplexModule::getStructureInfo(deque<VMModuleInfo>& s) const
1789. {
1790.  VMModuleInfo info;
1791.  info.name = getKey();
1792.  info.type = getType();
1793.  IModule* pStore = dynamic_cast<IModule*>(getModuleStore());
1794.  info.store = (pStore) ? pStore->getKey() : "NONE";
1795.  if (module_C) info.parts.push_back(module_C->getKey());
1796.  s.push_back(info);
1797. }
1798.
1799. bool ComplexModule::activateCurrentConf()
1800. {
1801.  // obter o almacén de módulos
1802.  IModuleStore* mstore = getModuleStore();
1803.  if (mstore == NULL) return false;
1804.  // obter o almacén de configuracións
1805.  IConfigurationStore* cstore = getConfigurationStore();
1806.  if (cstore == NULL) return false;
1807.  // obter a configuración actual
1808.  string conf_name = cstore->getCurrentConf();
1809.  const ModuleConfiguration& conf = cstore->getConfiguration(conf_name);
1810.  // obter un módulo tipo C
1811.  conf.getModuleNames("TYPE_C", names);
1812.  module_C = mstore->getModule("TYPE_C", names[0]);
1813. }
1814.
1815. bool ComplexModule::deactivateCurrentConf()
1816. {
1817.  module_C = NULL;
1818. }
```

```
1819. // Almacén de módulos
1820. class Store : public ModuleStore, public ConfigurationStore
1821. {
1822. public:
1823.  Store();
1824.  void initCategories();
1825.  void initConfigurations();
1826.  bool isValidConfiguration(const ModuleConfiguration& conf) const;
1827.  bool setDefaultConf();
1828. };
1829.
1830. Store::Store()
1831. {
1832.  initCategories();
1833.  initConfigurations();
1834.  setDefaultConf();
1835. }
1836.
1837. void Store::initCategories()
1838. {
1839.  addCategory("TYPE_A");
1840.  addCategory("TYPE_B");
1841.  addCategory("TYPE_C");
1842. }
1843.
1844. void Store::initConfigurations()
1845. {
1846.  // configuración A
1847.  ModuleConfiguration conf_A("CONF_A");
1848.  conf.addItem("TYPE_A", 1);
1849.  conf.addItem("TYPE_B", 1);
1850.  conf["TYPE_A"].addModule("MODULE_A");
1851.  conf["TYPE_B"].addModule("MODULE_B1");
1852.  addConfiguration(conf_A);
1853.  // configuración B
1854.  ModuleConfiguration conf_B("CONF_B");
1855.  conf.addItem("TYPE_A", 1);
1856.  conf.addItem("TYPE_B", 1);
1857.  conf["TYPE_A"].addModule("MODULE_A");
1858.  conf["TYPE_B"].addModule("MODULE_B2");
1859.  addConfiguration(conf_B);
1860. }
1861.
1862. bool Store::isValidConfiguration(const ModuleConfiguration& conf) const
1863. {
1864.  // comprobar que todos os elementos da configuración son válidos
1865.  deque<string> items;
1866.  conf.getItemNames(items);
1867.  for (deque<string>::const_iterator item = items.begin();
1868.         item != items.end(); ++item)
1869.     if (*item != "TYPE_A" && *item != "TYPE_B") return false;
1870.  return true;
1871. }
1872.
1873. bool Store::setDefaultConf()
1874. {
1875.  setCurrentConf("CONF_A");
1876. }
```

```
1877. // Módulo principal
1878. class MainModule : public StructuredModule, public ConfigurationStore, public Module
1879. {
1880.  // módulos agregados
1881.  IModule* module_A;
1882.  IModule* module_B;
1883. public:
1884.  MainModule(const string& n, IModuleStore* mstore, IConfigurationStore* cstore);
1885.  void getStructureInfo(deque<VMModuleInfo>& s) const;
1886.  bool activateCurrentConf();
1887.  bool deactivateCurrentConf();
1888.  void initConfigurations();
1889.  bool isValidConfiguration(const ModuleConfiguration& conf) const;
1890.  bool setDefaultConf();
1891. };
1892.
1893. MainModule::MainModule(const string& n, IModuleStore* mstore,
1894.                        IConfigurationStore* cstore)
1895.    : StructuredModule(mstore, cstore), Module(n, "MAIN")
1896. {
1897.  initConfigurations();
1898.  setDefaultConf();
1899. }
1900.
1901. void MainModule::getStructureInfo(deque<VMModuleInfo>& s) const
1902. {
1903.  VMModuleInfo info;
1904.  info.name = getKey();
1905.  info.type = getType();
1906.  IModule* pStore = dynamic_cast<IModule*>(getModuleStore());
1907.  info.store = (pStore) ? pStore->getKey() : "NONE";
1908.  if (module_A) info.parts.push_back(module_A->getKey());
1909.  if (module_B) info.parts.push_back(module_B->getKey());
1910.  s.push_back(info);
1911.  // obter información dos módulos complexos agregados
1912.  if (module_A) module_A->getStructureInfo(s);
1913. }
1914.
1915. bool MainModule::activateCurrentConf()
1916. {
1917.  // obter almacén de módulos
1918.  IModuleStore* mstore = getModuleStore();
1919.  if (mstore == NULL) return false;
1920.  // obter almacén de configuracións
1921.  IConfigurationStore* cstore = getConfigurationStore();
1922.  if (cstore == NULL) return false;
1923.  // obter configuración actual
1924.  string conf_name = cstore->getCurrentConf();
1925.  const ModuleConfiguration& conf = cstore->getConfiguration(conf_name);
1926.  // obter módulo tipo A
1927.  deque<string> names;
1928.  conf.getModuleNames("TYPE_A", names);
1929.  module_A = mstore->getModule("TYPE_A", names[0]);
1930.  // activar configuración do módulo A
1931.  module_A->setModuleStore(mstore);
1932.  module_A->setConfigurationStore(this);
1933.  module_A->activateCurrentConf();
1934.  // obter módulo tipo B
1935.  conf.getModuleNames("TYPE_B", names);
1936.  module_B = mstore->getModule("TYPE_B", names[0]);
1937. }
```

```
1938. bool MainModule::deactivateCurrentConf()
1939. {
1940.  module_A = NULL;
1941.  module_B = NULL;
1942. }
1943.
1944. void MainModule::initConfigurations()
1945. {
1946.  // configuración A
1947.  ModuleConfiguration conf_A("CONF_A");
1948.  conf.addItem("TYPE_C", 1);
1949.  conf["TYPE_C"].addModule("MODULE_C1");
1950.  addConfiguration(conf_A);
1951.  // configuración B
1952.  ModuleConfiguration conf_B("CONF_B");
1953.  conf.addItem("TYPE_C", 1);
1954.  conf["TYPE_C"].addModule("MODULE_C2");
1955.  addConfiguration(conf_B);
1956. }
1957.
1958. bool MainModule::isValidConfiguration(const ModuleConfiguration& conf) const
1959. {
1960.  // comprobar que todos os módulos da configuración son válidos
1961.  deque<string> items;
1962.  conf.getItemNames(items);
1963.  for (deque<string>::const_iterator item = items.begin();
1964.         item != items.end(); ++item)
1965.    if (*item != "TYPE_C") return false;
1966.  return true;
1967. }
1968.
1969. bool MainModule::setDefaultConf()
1970. {
1971.  setCurrentConf("CONF_A");
1972. }
1973.
1974. // Construcción da arquitectura
1975. int main()
1976. {
1977.  // creación dos módulos dinámicos
1978.  IModule* module_A = dynamic_cast<IModule*>(new ComplexModule("MODULE_A", "TYPE_A"));
1979.  IModule* module_B1 = dynamic_cast<IModule*>(new Module("MODULE_B1", "TYPE_B"));
1980.  IModule* module_B2 = dynamic_cast<IModule*>(new Module("MODULE_B2", "TYPE_B"));
1981.  IModule* module_C1 = dynamic_cast<IModule*>(new Module("MODULE_C1", "TYPE_C"));
1982.  IModule* module_C2 = dynamic_cast<IModule*>(new Module("MODULE_C2", "TYPE_C"));
1983.  // almacenamento dos módulos
1984.  Store store;
1985.  store.addModule(module_A);
1986.  store.addModule(module_B1);
1987.  store.addModule(module_B2);
1988.  store.addModule(module_C1);
1989.  store.addModule(module_C2);
1990.  // creación e activación do módulo principal
1991.  MainModule main_module("MAIN", &store, &store);
1992.  main_module.activateCurrentConf(); // actívase CONF_A
1993.  return 0;
1994. }
```

Os módulos simples son implementados mediante instancias da clase *Module* (liñas 1762-1770), derivada da interface *IModule* (§7.1.1.1), que declara atributos para almacenar o nome e tipo do módulo e implementa os métodos de acceso aos valores deses atributos. Os módulos complexos de tipo A son implementados mediante instancias da clase *ComplexModule* (liñas 1773-1782), que declara un atributo (liña 1776) para referenciar ao módulo simple de tipo C que forma parte da súa arquitectura. A activación da configuración actual do módulo complexo

impleméntase no método *activateCurrentConf* (liñas 1799-1813), no que se obtén a información da configuración actual do almacén de configuracións (liña 1809) e utilízase para obter do almacén de módulos a referencia ao módulo agregado (liña 1812). A implementación da clase *ComplexModule* complétase cos métodos *getStructureInfo* (liñas 1788-1797), que devolve unha representación da estructura actual do módulo, e *deactivateCurrentConf* (liñas 1815-1818), que anula o apuntador ao valor do módulo agregado.

Os módulos son almacenados nunha instancia da clase *Store* (liñas 1820-1828), derivada da clase *ModuleStore* (§7.1.1.3), organizados en tres categorías que son iniciadas na implementación do método *initCategories* (liñas 1837-1842). Ademais este módulo tamén almacena as configuracións válidas para o módulo *main*, que son iniciadas no método *initConfigurations* (liñas 1844-1860). A clase implementa o método *isValidConfiguration* (liñas 1862-1871), que neste exemplo comproba para unha configuración dada se os módulos que a forman son dos tipos agardados.

O módulo principal da arquitectura é unha instancia da clase *MainModule* (liña 1878-1891), que declara dous atributos para referenciar os módulos agregados que forman a súa estructura. Ademais este módulo tamén serve como almacén das configuracións válidas para os módulos complexos de tipo A. As configuracións válidas son iniciadas na implementación do método *initConfigurations* (liñas 1944-1956) e a validez dunha configuración comprobada polo método *isValidConfiguration* (liñas 1958-1967). A activación da configuración actual é implementada no método *activateCurrentConf* (liñas 1915-1937), no que se obteñen do módulo *store*, que é o que fai as veces de almacén de módulos e configuracións do módulo principal, tanto a información da configuración actual (liña 1925) como as referencias aos módulos agregados (liñas 1929 e 1936). Nótese que este módulo é o responsábel tamén de activar a configuración dos módulos complexos que forman a súa estructura (liñas 1931-1933). A implementación da clase *MainModule* complétase cos métodos *getStructureInfo* (liñas 1901-1913), que devolve unha representación da estructura actual do módulo, e *deactivateCurrentConf* (liñas 1938-1942), que anula os valores dos apuntadores aos módulos agregados. Nótese que o método *getStructureInfo* aplícase recursivamente ao módulo complexo agregado de tipo A (liña 1912).

7.1.2. Operativa da arquitectura: os procesos

Dende o punto de vista operativo, a arquitectura interna da máquina virtual está formada por múltiples procesos concorrentes que se comunican tanto síncrona como asincronamente. Aínda que se lles denomine igual, non é o mesmo un proceso da máquina virtual que un proceso do sistema operativo sobre o que esta se execute. No deseño e implementación da máquina virtual tentouse que as dependencias dos aspectos específicos de cada sistema operativo fosen mínimas para facilitar a portabilidade. O concepto de proceso utilizado na máquina virtual é a abstracción lóxica dunha secuencia de execución e proporciona as funcións necesarias para controlar o seu estado e a comunicación coas demais secuencias da máquina virtual.

A implementación dun proceso da máquina virtual pode realizarse de diferentes formas, por exemplo asignándoo a un ou máis procesos, tarefas ou "threads" do sistema operativo. A implementación actual da máquina virtual presupón a existencia dun sistema operativo "multithread", no que a máquina é executada nun proceso do sistema operativo e cada proceso da máquina virtual é executado nun "thread" do sistema operativo. En consecuencia, a comunicación entre procesos da máquina virtual é implementada utilizando os mecanismos de comunicación entre "threads" do sistema operativo. No resto deste apartado descríbense as funcionalidades proporcionadas polos procesos da máquina virtual, explícase como foron definidas e implementadas e móstranse exemplos da súa utilización.

7.1.2.1. As funcionalidades dun proceso

Os procesos da máquina virtual proporcionan catro funcionalidades básicas:

1. O control do estado de execución do proceso.
2. A exclusión mutua no acceso aos datos internos do proceso.
3. A comunicación asíncrona entre procesos mediante paso de mensaxes.
4. A notificación da finalización de operacións asíncronas.

A interface abstracta dun proceso inclúe as declaracións dos métodos a redefinir nas clases derivadas para implementar as tres primeiras funcionalidades, mentres que a transferencia do fluxo de control cando se produce unha notificación asíncrona é unha característica implementada internamente nas subclases.

O código da interface abstracta dun proceso é o seguinte:

```
1995.class IVMProcess
1996.{
1997.public:
1998.  // control do estado do proceso
1999.  virtual void Start() = 0;
2000.  virtual void Suspend() = 0;
2001.  virtual void Resume() = 0;
2002.  virtual void Finish() = 0;
2003.  virtual bool isSuspended() = 0;
2004.  virtual bool isRunning() = 0;
2005.protected:
2006.  // exclusión mutua no acceso aos datos internos
2007.  virtual bool tryDataAccess() = 0;
2008.  virtual void lockDataAccess() = 0;
2009.  virtual void unlockDataAccess() = 0;
2010.public:
2011.  // xestión de conexións
2012.  virtual bool isValidPort(const string& port, bool& state) = 0;
2013.  virtual bool isConnected(const string& port, bool& state) = 0;
2014.  virtual bool connect(const string& port, VMChannel* channel) = 0;
2015.  virtual bool disconnect(const string& port) = 0;
2016.  virtual bool getConnection(const string& port, VMChannel* channel) = 0;
2017.protected:
2018.  // lectura/escritura de mensaxes
2019.  virtual bool read(const string& port, IVMMsg** msg) = 0;
2020.  virtual bool read(const string& port,
2021.                    IVMMsg** msg,
2022.                    pfn callbck,
2023.                    IAsyncClbkArg* clbk_arg = NULL) = 0;
2024.  virtual bool write(const string& port, IVMMsg* msg) = 0;
2025.protected:
2026.  // código do proceso
2027.  virtual void Initial() = 0;
2028.  virtual bool Run() = 0;
2029.  virtual void Final() = 0;
2030.  virtual bool OnSuspend() = 0;
2031.  virtual bool OnResume() = 0;
2032.  virtual bool OnEvent() = 0;
2033.  virtual bool OnHandler(void* arg) = 0;
2034.  virtual bool OnError() = 0;
2035.};
```

A interface declara métodos para controlar e consultar o estado de execución dun proceso (liñas 1999-2004), garantir a exclusión mutua no acceso aos datos internos (liñas 2007-2009), configurar as conexións para o paso de mensaxes entre procesos (liñas 2012-2016) e escribir e ler mensaxes (liñas 2019-2024). Nótese que parte dos métodos declarados son *protected*, polo que unicamente poden ser utilizados na implementación das clases derivadas. O código específico de cada proceso é implementado redefinindo os métodos declarados nas liñas 2027-2034 nas clases derivadas. Estes métodos son executados en diferentes estados do proceso segundo se explica no apartado seguinte.

7.1.2.2. O ciclo de vida dun proceso

A Figura 7.4 mostra o diagrama de estados do ciclo de vida dun proceso da máquina virtual. Durante este ciclo cada proceso ten unha prioridade que lle é asignada na súa creación e non pode ser modificada. As transicións entre estados correspóndense con eventos producidos por chamadas aos métodos da interface do proceso: *Start* (liña 1999), *Suspend* (liña 2000), *Resume* (liña 2001) e *Finish* (liña 2002); ou por condicións internas detectadas durante a execución: *error*, *async_call* e *sync_call*. As accións executadas nas transicións do diagrama correspóndense cos métodos que conteñen o código específico de cada proceso declarados na interface *IVMProcess* (liñas 2027-2034).

7.1.2.3. A implementación dos procesos

A clase *VMProcess* proporciona a implementación por defecto da interface *IVMProcess* (§7.1.2.1) supoñendo que cada unha das súas instancias vai ser asignada a un "thread" do sistema operativo. Ademais esta clase implementa a interface *IModule* (§7.1.1.1), polo que proporciona tamén a funcionalidade precisa para cargar e descargar procesos dinamicamente na memoria da máquina virtual. A lóxica do ciclo de vida da Figura 7.4 dende que o proceso é iniciado invocando o método *Start* (liña 1999), ata que a súa execución remata pola invocación do método *Finish* (liña 2002) ou pola detección dun erro, é implementada no método *Run* (liña 2028) da clase *VMProcess*.

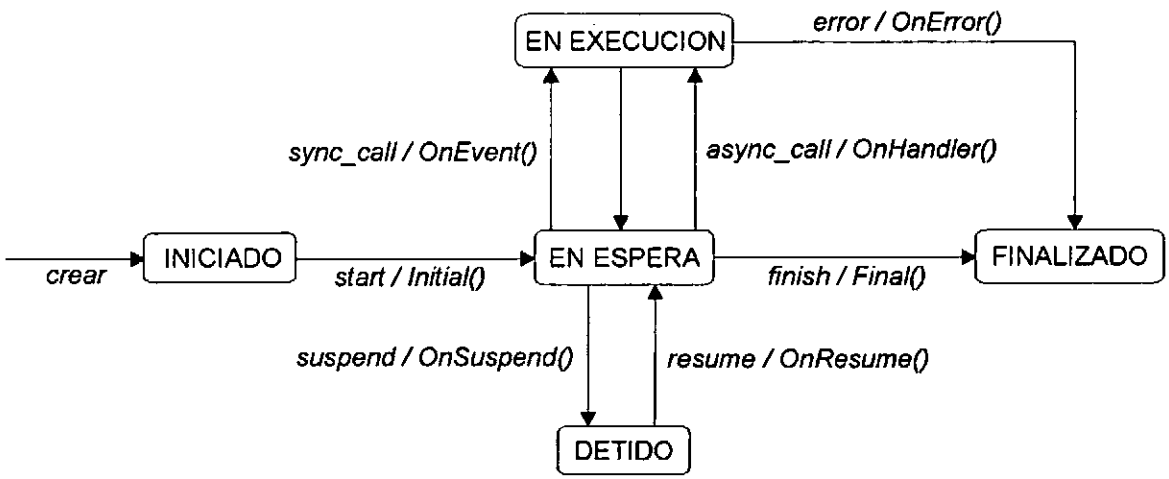


Figura 7.4. Diagrama de estados do ciclo de vida dun proceso da máquina virtual.

O código seguinte mostra a implementación deste método:

```
2036. void SFCVMProcess::Run()
2037. {
2038.   while (!exit)
2039.   {
2040.     unsigned result = getEvent()->Wait(); // bloqueo do proceso
2041.     switch (result)
2042.     {
2043.     case IAlertableEvent::AE_SIGNALED: // proceso desbloqueado
2044.          getEvent()->Reset();
2045.          if (!exit)
2046.          {
2047.             if (suspend)
2048.             {
2049.               // invocouse o método suspend
2050.               exit |= !OnSuspend();
2051.               if (!exit)
2052.               {
2053.                    --ssuspend; // agardar invocación do método resume
2054.                    suspend = false;
2055.                    exit |= !OnResume();
2056.               }
2057.             }
2058.             else if (clbk)
2059.             {
2060.               // recibiuse unha notificación asíncrona
2061.               exit |= !OnHandler(clbk_arg);
2062.               clbk = false;
2063.               clbk_arg = NULL;
2064.             }
2065.             else
2066.               // invocouse un método do proceso
2067.               exit |= !OnEvent();
2068.          }
2069.       break;
2070.       default:
2071.           exit |= !OnError(); // erro
2072.       break;
2073.     }
2074.   }
2075. }
```

Unha vez iniciado un proceso, a súa execución realízase nun bucle do que non se sae ata que a variábel booleana *exit* é activada, xa sexa porque se invoque o método *Finish* (liña 2002) ou debido á detección dalgunha condición interna na execución dos métodos asociados ás transicións do diagrama de estados da Figura 7.4 (liñas 2030-2034). En cada iteración do bucle a execución do proceso bloquease (liña 2040) ata a ocorrencia dalgún dos eventos seguintes:

1. A invocación dende outro proceso dalgún dos métodos de control do estado (liñas 2000-2002): *Finish*, *Suspend* ou *Resume*.
2. A notificación dende outro proceso da finalización dunha operación asíncrona iniciada por este proceso.
3. O inicio dende outro proceso dunha operación asíncrona que este proceso implemente.

No resto deste apartado explícanse os detalles da implementación do mecanismo de bloqueo da execución do proceso e os eventos asociados á invocación dos seus métodos. As notificacións asíncronas son explicadas en (§7.1.2.4.2).

7.1.2.3.1. O bloqueo da execución dun proceso

O bloqueo do proceso é implementado utilizando un tipo de semáforo cuxa interface abstracta é a seguinte:

```
2076. struct IAlertableEvent
2077. {
2078.  // resultado dunha solicitude de bloqueo
2079.  enum {AE_ERROR = 0, AE_SIGNALED, AE_TIME_OUT};
2080.  // bloqueo, desbloqueo, consulta e iniciación
2081.  virtual unsigned Wait() = 0;
2082.  virtual unsigned Wait(unsigned timer) = 0;
2083.  virtual void Signal() = 0;
2084.  virtual bool isSignaled() = 0;
2085.  virtual void Reset() = 0;
2086. };
```

Esta interface declara dúas versións do método *Wait* (liñas 2081 e 2082), que permiten bloquear un proceso indefinidamente ou durante un período de tempo determinado. O proceso estará bloqueado ata que transcorra o período de tempo indicado ou outro proceso invoque o método *Signal* (liña 2083). O método *Reset* (liña 2085) sirve para reiniciar o semáforo e permitir novos bloqueos. Cada proceso crea e almacena durante a súa iniciación unha instancia que implemente esta interface[70], que é accedida utilizando o método privado *getEvent* (liña 2040) da clase *VMProcess*.

7.1.2.3.2. Invocación dos métodos *Suspend* e *Resume* dende outro proceso

Na versión "multithread" da máquina virtual, a suspensión e reinicio dun proceso é implementada nos métodos *Suspend* (liña 2087-2098) e *Resume* (liña 2100-2111) da clase *VMProcess*. A utilización destes métodos presenta as restricción seguintes[71]:

1. Un proceso non pode ser suspendido simultaneamente por varios procesos.
2. Un proceso só pode ser reiniciado polo proceso que o suspenda.

Estas restricción son garantidas internamente na implementación dos métodos *Suspend* e *Resume* utilizando os seguintes atributos privados da clase *VMProcess*:

1. *suspend*, un indicador booleano que indica se o proceso está detido.
2. *ssuspend*, un semáforo no que se bloquea a execución do proceso mentres está detido.
3. *msuspend*, un "mutex" para implementar a segunda das restricción comentadas anteriormente cando varios procesos invocan simultaneamente o método *Suspend*. Este "mutex" proporciona un mecanismo de exclusión mutua de xeito que o primeiro proceso que obteña o "mutex" será o que teña o control do reinicio do proceso suspendido.

---

[70] A versión Windows da máquina virtual utiliza o mecanismo de sincronización entre "threads" denominado *Event Object* para implementar esta interface.

[71] A utilización incorrecta dos métodos *Suspend* e *Resume* pode provocar o bloqueo indefinido dun proceso. A versión de depuración da implementación destes métodos emite mensaxes de aviso para os erros mais comúns sendo responsabilidade do programador garantir a súa utilización correcta.

O código seguinte mostra unha versión simplificada dos métodos *Suspend* e *Resume*:

```
2087. void VMProcess::Suspend(void)
2088. {
2089.   if (msuspend.TryEnterMutex()) // obter "mutex"
2090.   {
2091.     if (suspend)
2092.       // indicar que xa estaba detido (depuración)
2093.     suspend = true;          // activar indicador
2094.     getEvent()->Signal(); // desbloquear proceso
2095.   }
2096.   else
2097.     // indicar que xa foi detido por outro proceso (depuración)
2098. }
2099.
2100. void VMProcess::Resume()
2101. {
2102.   try
2103.   {
2104.     msuspend.LeaveMutex();  // liberar "mutex"
2105.     ++ssuspend;             // incrementar semáforo (=> resume o proceso)
2106.   }
2107.   catch (...)
2108.   {
2109.     // indicar que non pode ser reiniciado (depuración)
2110.   }
2111. }
```

Cando un proceso invoca o método *Suspend* tenta obter o "mutex" *msuspend* (liña 2089). Se este está ocupado o proceso xa foi suspendido previamente por outro proceso, polo que a execución continua sen modificar o estado do proceso suspendido. En caso contrario, actívase o indicador *suspend* (liña 2093) e desbloquease o proceso de xeito que a operación conclúa no método *Run*, executando o método *OnSuspend* (liña 2050) no que se suspende o proceso bloqueándoo no semáforo *ssuspend* (liña 2053). Para reiniciar a execución do proceso invócase o método *Resume*, no que se libera o mutex *msuspend* (liña 2104) e se desbloquea o proceso suspendido incrementando o semáforo *ssuspend* (liña 2105). A operación complétase no método *Run*, no que se desactiva o indicador *suspend* (liña 2054), execútase o método *OnResume* (liña 2055) e continúase coa execución do proceso. En caso de que o proceso que invoque o método *Resume* non sexa o mesmo que detivo o proceso suspendido, o intento de liberar o mutex *msuspend* provocará unha excepción e a execución continuará sen modificarse o estado do proceso suspendido.

7.1.2.3.3. Invocación do método *Finish* dende outro proceso

O código seguinte mostra unha versión simplificada do método *Finish* (liña 2002):

```
2112. void VMProcess::Finish(void)
2113. {
2114.   exit = true;
2115.   if (suspend)
2116.     Resume();
2117.   else
2118.     getEvent()->Signal();
2119. }
```

A execución deste método activa o indicador *exit* (liña 2114) para que o proceso saia do bucle do método *Run* e, dependendo do estado do proceso —valor do indicador *suspend* (liña 2115)—, reiníciase a súa execución ou desbloquease. Debido ás restriccións comentadas anteriormente, un proceso suspendido unicamente pode ser finalizado polo proceso que o suspendeu.

7.1.2.3.4. Inicio dunha operación asíncrona dende outro proceso

O seguinte código mostra un método xenérico que implemente o inicio dunha operación asíncrona implementada polo proceso:

```
2120. void AnyVMProcess::AnyMethod(void)
2121. {
2122.  lockDataAccess();
2123.  // código específico do método (modifica datos internos se é preciso)
2124.  getEvent()->Signal(); // desbloquear proceso
2125.  unlockDataAccess();
2126. }
```

A implementación deste método comeza obtendo o "mutex" (liña 2122) que regula o acceso aos datos internos do proceso (a execución detense ata que o "mutex" estea libre). Unha vez que se obtén o "mutex" realízanse as operacións que preparan a execución asíncrona (liña 2123), modificando os datos internos do proceso se é preciso, desbloquease o proceso (liña 2124) e finalmente libérase o "mutex" (liña 2125). A operación complétase no método *Run*, invocando o método *OnEvent* (liña 2067) que implementa o código da operación asíncrona que é específico de cada proceso.

7.1.2.4. A comunicación entre procesos

A comunicación entre procesos da máquina virtual pode realizarse mediante unha das tres técnicas seguintes:

1. Como os procesos da máquina virtual son instancias de clases derivadas da interface *IVMProcess* (§7.1.2.1), o mecanismo de comunicación máis simple consiste na invocación dende un proceso dalgún dos métodos implementados por outro. O uso desta técnica require que o proceso 'invocador' poda obter unha referencia válida ao proceso 'invocado', o que en ocasións implica coñecer de antemán o tipo concreto deste proceso. Exemplos do uso desta técnica son as invocacións dos métodos *Finish*, *Suspend* ou *Resume* explicados en (§7.1.2.3.2; §7.1.2.3.3).
2. En ocasións a invocación dun método nun proceso sirve para iniciar unha operación asíncrona (§7.1.2.3.4). Existen situacións nas que o proceso 'invocador' require dunha notificación que indique cando esta remata. O segundo mecanismo de comunicación entre procesos consiste na notificación da finalización de operación asíncronas, tal e como se explica en (§7.1.2.4.2).
3. A terceira técnica consiste no envío de mensaxes asíncronas entre procesos. As mensaxes poden ser utilizadas, por exemplo, para intercambiar información, iniciar operacións asíncronas ou notificar a ocorrencia de eventos. Para que dous procesos podan intercambiar mensaxes é preciso que exista entre eles unha *conexión*. As conexións son unidireccionais e poden restrinxir o tipo de mensaxes transmitidas a través delas. En cada proceso se indica o número e tipo de conexións que poden realizarse mediante a declaración de *portas de enlace*. Unha conexión entre dous procesos crease unindo dúas portas de enlace do mesmo tipo (unha en cada proceso) mediante un *canal de comunicacións*, que fai as veces de medio de transmisión entre os procesos. Esta técnica ten a avantaxe de que o acoplamento entre os procesos é moi débil ao non existir referencias directas entre eles o que facilita a reconfiguración dinámica da arquitectura da máquina virtual mediante a modificación das conexións entre procesos.

No resto deste apartado explícanse en detalle a implementación dos mecanismos de paso de mensaxes e de notificación asíncrona entre procesos da máquina virtual na súa versión "multithread".

7.1.2.4.1. Paso de mensaxes entre procesos

Na Figura 7.5 móstrase o diagrama das clases utilizadas na implementación do mecanismo de paso de mensaxes entre procesos. Como pode verse son catro as clases principais implicadas na implementación deste mecanismo:

1. A clase *VMProcess*, derivada de *IVMProcess* (§7.1.2.1), proporciona a implementación por defecto dos métodos relacionados coa conexión de procesos (liñas 2012-2016) e o envío e recepción de mensaxes (liñas 2019-2024). Esta clase tamén almacena o conxunto de portas de enlace do proceso —agregación *ports*—.
2. A clase *VMPort*, especialización da clase parametrizada *Port* para mensaxes de tipo *IVMMsg*, representa as portas de enlace utilizadas para realizar as conexións entre procesos. Cada porta de enlace ten un identificador alfanumérico —atributo *id*— único no proceso que a conteña e mantén unha referencia —agregación *channel*— ao canal utilizado como medio de transmisión da conexión. A clase parametrizada *Port* declara métodos para conectar a porta a un canal dado e consultar o seu estado. Nótese que poden crearse portas de distintos tipos que restrinxan as mensaxes transmitidas a través delas mediante diferentes especializacións da clase parametrizada *Port*.
3. A clase *VMChannel*, especialización da clase parametrizada *Channel* para mensaxes de tipo *IVMMsg*, representa o medio de transmisión utilizado na conexión entre procesos. Os canais implementan comunicacións N a 1. Cada canal ten a responsabilidade de almacenar as mensaxes —agregación *buffer*— na orde na que son recibidas mentres non sexan recuperadas polo proceso receptor. A clase parametrizada *Channel* declara métodos para ler e escribir mensaxes no canal e, do mesmo xeito que coas portas de enlace, poden crearse canais de diferentes tipos que restrinxan as mensaxes transmitidas a través deles. Na versión actual da implementación da clase parametrizada *Channel* garántese a exclusión mutua no acceso simultáneo ao canal e non se tratan os problemas relacionados co desbordamento do "buffer" de mensaxes.
4. A clase *IVMMsg*, interface abstracta da que derivar as clases de mensaxes intercambiadas entre os procesos da máquina virtual.

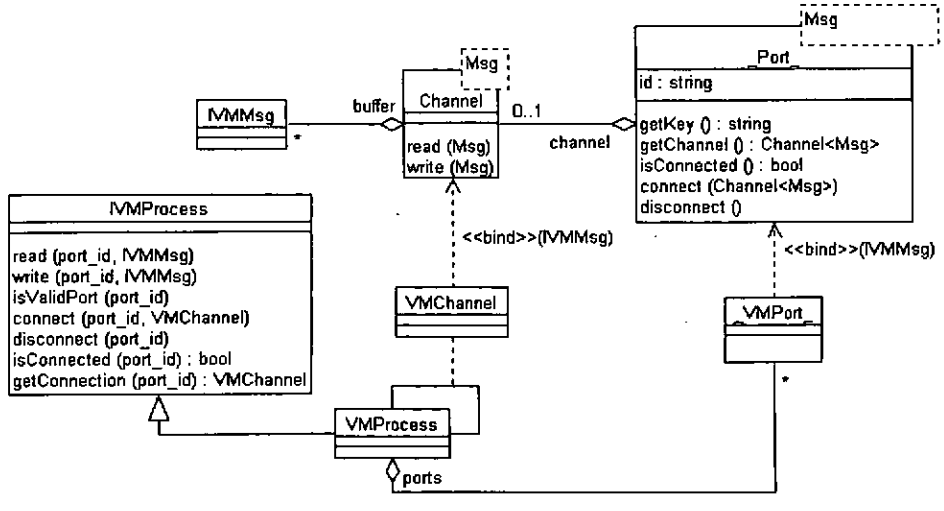


Figura 7.5. Diagrama das clases que modelan o mecanismo de paso de mensaxes entre procesos.

O método *read* da clase *VMChannel*, utilizado polo proceso receptor para recuperar as mensaxes almacenadas nun canal, implementouse con dúas semánticas diferentes:

1. *Con bloqueo*, o proceso receptor detén a súa execución ata que haxa algunha mensaxe dispoñíbel no canal.
2. *Sen bloqueo*, utilízase o mecanismo de notificación asíncrona para notificarlle ao proceso receptor a presencia dunha mensaxe no canal.

O diagrama da Figura 7.6 mostra a interacción entre os obxectos implicados no envío dunha mensaxe dende un proceso da máquina virtual a outro. A primeira secuencia mostra o envío da mensaxe, que é almacenada no canal, mentres que a segunda secuencia mostra a lectura con bloqueo. Na implementación do canal utilízase un "mutex" para controlar o acceso ao "buffer" de mensaxes e un semáforo para bloquear a execución do proceso receptor ata que haxa algunha mensaxe dispoñíbel.

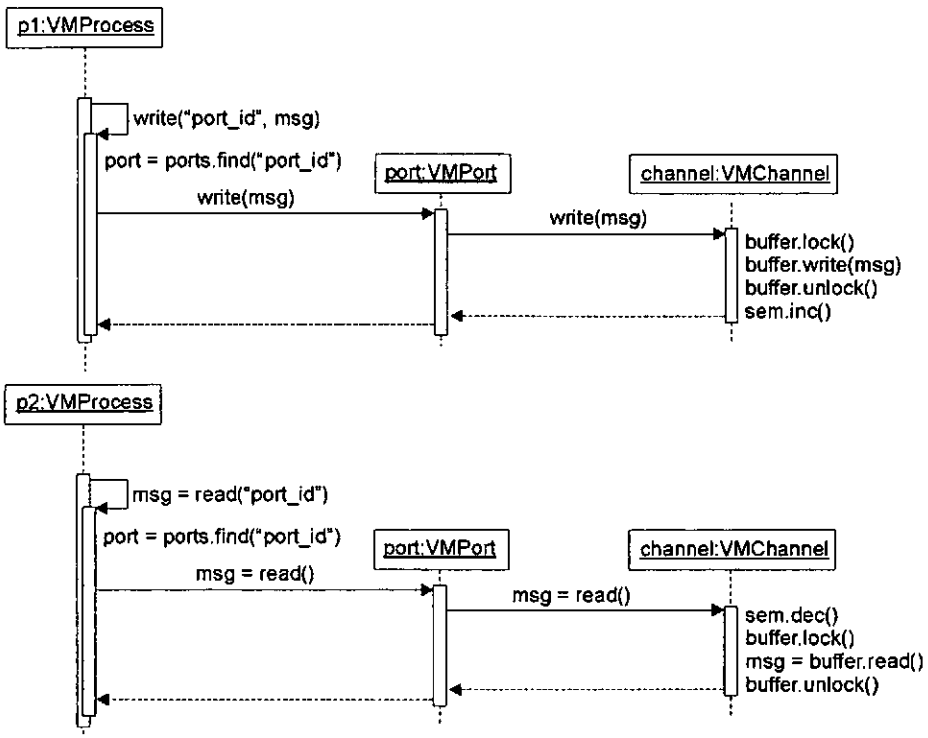


Figura 7.6. Secuencias de mensaxes intercambiadas no envío e recepción dunha mensaxe.

7.1.2.4.2. Notificación da finalización de operacións asíncronas

Existen situacións nas que dende un proceso se inicia unha operación asíncrona noutro e, unha vez que esta se completa, continúase coa execución doutras operacións. A implementación deste funcionamento require dalgún mecanismo que permita dende un proceso notificar a outro a finalización dunha operación asíncrona. Nótese que a técnica de paso de mensaxes explicada no apartado anterior pode ser utilizada para iniciar operacións asíncronas e notificar a súa finalización naqueles casos nos que os procesos implicados estean conectados mediante un canal e non haxa requirimentos relacionados co tempo que a notificación pase almacenada neste antes de ser procesada. Sen embargo estas condicións non sempre se cumpren, polo que se implementou na máquina virtual unha técnica que permite realizar notificacións asíncronas inmediatas entre procesos non conectados mediante un canal.

A técnica basease na utilización de funcións "callback" e presupón que os procesos implicados na comunicación pertencen ao mesmo espacio de direccionamento. A súa implementación consiste en almacenar, no proceso que realiza a operación asíncrona, unha referencia a unha función que é executada cando a operación remata. Esta función notifica a finalización da operación ao proceso que a iniciou e, opcionalmente, pásalle un argumento almacenado xunto coa función. A clase *AsyncClbkContainer* implementa o comportamento por defecto dun proceso que realice notificacións asíncronas. O seguinte código mostra a implementación desta clase:

```
2127. class AsyncClbkContainer
2128. {
2129. private:
2130.   pfn async_fun;
2131.   IAsyncClbkArg* async_fun_arg;
2132. public:
2133.   // consulta, modificación e execución da notificación
2134.   pfn getCallback(void) { return async_fun; }
2135.   IAsyncClbkArg* getCallbackArg(void) { return async_fun_arg; }
2136.   void setCallback(pfn fun, IAsyncClbkArg* fun_arg = NULL)
2137.   {
2138.     async_fun = fun;
2139.     async_fun_arg = fun_arg;
2140.   }
2141.   void callCallback(void) const
2142.   {
2143.     if (async_fun)
2144.       (*async_fun)(async_fun_arg);
2145.   }
2146. };
2147. // declaración do tipo da función "callback"
2148. typedef void (*pfn) (IAsyncClbkArg*);
2149. // argumento da notificación
2150. struct IAsyncClbkArg
2151. {
2152.   virtual IAsyncClbkArg* clone() = 0;
2153. };
```

A clase *AsyncClbkContainer* almacena as referencias á función "callback" e ao argumento opcional nos atributos *async_fun* (liña 2130) e *async_fun_arg* (liña 2131), respectivamente. Tamén declara métodos para consultar (liñas 2134-2135) e almacenar (liñas 2136-2140) a función "callback" e o seu argumento, e para executar a notificación asíncrona (liñas 2141-2145). Como pode verse na liña 2148, suponse que a función "callback" non devolve ningún resultado e ten un argumento de tipo *IAsyncClbkArg*, que é a interface abstracta (liñas 2150-2153) da que derivan todas as clases utilizadas como argumento dunha notificación asíncrona. Esta implementación ten dúas restriccións:

1. As funcións "callback" só poden ser funcións globais ou métodos estáticos dunha clase[72].
2. Cada proceso non pode almacenar máis dunha función "callback" simultaneamente. Isto limita o número de operacións asíncronas simultáneas que un proceso pode notificar[73].

A recepción dunha notificación asíncrona é implementada na clase *VMProcess*, que proporciona un mecanismo xenérico reutilizado en todas as clases derivadas. Esta clase declara

[72] Pode consultarse [142] para unha explicación detallada de porqué non poden utilizarse os métodos non estáticos dunha clase como funcións "callback".

[73] Ainda que podería realizarse unha implementación que eliminara esta restricción, na versión actúal da máquina virtual nunca se utilizou máis dunha notificación asíncrona por proceso.

un método estático, que será utilizado como función "callback" por todos os procesos, e dous atributos privados auxiliares. O código seguinte mostra as declaracións e a implementación do método estático na clase *VMProcess*:

```
2154. class VMProcess
2155. {
2156.  bool clbk;      // indicador da recepción dunha notificación asíncrona
2157.  void* clbk_arg; // argumento da notificación
2158. protected:
2159.  static void async_callback(IAsyncClbkArg* arg); // función "callback"
2160. };
2161. // recepción dunha notificación asíncrona
2162. void VMProcess::async_callback(IAsyncClbkArg* arg)
2163. {
2164.  async_clbk_arg* parg = dynamic_cast<async_clbk_arg*>(arg);
2165.  if (parg)
2166.  {
2167.   parg->pthis->clbk = true;           // activar indicador
2168.   parg->pthis->clbk_arg = parg->parg; // almacenar argumento
2169.   parg->pthis->getEvent()->Signal();  // desbloquear proceso
2170.  }
2171.  delete arg;
2172. }
```

O método *async_callback* (liña 2159) presupón que recibe unha instancia da clase *async_clbk_arg*. Esta clase é derivada de *IAsyncClbkArg* (liñas 2150-2153) e almacena unha referencia ao proceso que iniciou a operación asíncrona e un argumento opcional que lle será enviado ao proceso cando a operación remate. A implementación do método utiliza a información almacenada nesta instancia para acceder ao proceso que iniciou a operación asíncrona (liña 2164), activar o valor do atributo *clbk* (liña 2167), almacenar a información opcional da notificación no atributo *clbk_arg* (liña 2168) e desbloquear o proceso (liña 2169). O procesamento da notificación complétase no método *Run* no que se invoca o método *OnHandler* (liña 2061), pasándolle como argumento a información almacenada no atributo *clbk_arg*. Finalmente reinícianse os valores dos atributos *clbk* e *clbk_arg* (liñas 2062 e 2063) para permitir unha nova notificación.

Unha limitación da implementación descrita é que unicamente se almacena a última notificación recibida por un proceso, polo que cando un proceso suspendido que teña unha notificación pendente reciba unha nova notificación, a pendente pérdese. Esta situación repetirase mentres o proceso suspendido continúe recibindo notificacións. É responsabilidade do programador ter en conta esta limitación naquelas situacións nas que sexa preciso procesar todas as notificacións asíncronas. No seguinte exemplo móstrase o código mínimo preciso para implementar a solicitude dun servicio asíncrono con notificación de finalización entre un proceso cliente e un proceso servidor utilizando a técnica explicada previamente:

```
2173. // Proceso servidor
2174. class Server : public VMProcess, public AsyncClbkContainer
2175. {
2176.  bool OnEvent();
2177. public:
2178.  void initAsyncService(pfn clbk, IAsyncClbkArg* clbk_arg);
2179. };
2180.
2181. bool Server::OnEvent()
2182. {
2183.  // executar aquí o código do servicio
2184.  callCallback(); // invocar función "callback"
2185. }
```

```
2186. void Server::initAsyncService(pfn clbk, IAsyncClbkArg* clbk_arg)
2187. {
2188.  lockDataAcess();
2189.  setCallback(clbk, clbk_arg);// almacenar información "callback"
2190.  getEvent()->Signal();       // desbloquear proceso
2191.  unlockDataAcess();
2192. }
2193.
2194. // Proceso cliente
2195. class Client : public VMProcess
2196. {
2197.  Server* pServer;
2198.  void requestAsyncService();
2199.  bool OnEvent();
2200.  bool OnHandler(void* arg);
2201. public:
2202.  Client(Server* srv) : pServer(srv) {}
2203.  void Operation();
2204. };
2205.
2206. bool Client::OnEvent()
2207. {
2208.  // executar aquí o código previo ao inicio do servicio asíncrono
2209.  requestAsyncService (); // iniciar servicio asíncrono
2210.  // executar aquí o código posterior ao inicio do servicio asíncrono
2211. }
2212.
2213. bool Client::OnHandler(void* arg)
2214. {
2215.  // executar aquí o código de manexo da notificación asíncrona
2216. }
2217.
2218. void Client::requestAsyncService()
2219. {
2220.  async_clbk_arg* parg = new async_clbk_arg(this);
2221.  if (parg)
2222.   pServer->initAsyncService(async_callback, parg); // iniciar servicio
2223. }
2224.
2225. void Client::Operation()
2226. {
2227.  lockDataAcess();
2228.  // modificar aquí os datos internos do cliente
2229.  getEvent()->Signal(); // desbloquear proceso
2230.  unlockDataAcess();
2231. }
2232.
2233. // Execución do exemplo
2234. int main()
2235. {
2236.  Server server;
2237.  Client client(&server);
2238.  client.Operation();
2239. }
```

Na implementación da solicitude da operación asíncrona (liñas 2218-2223), ao servidor pásanselle como argumentos unha referencia ao método estático *async_callback* e outra ao proceso que solicita o inicio da operación (mediante o apuntador *this* do C++). No procesamento da solicitude (liñas 2186-2192) o servidor almacena estes argumentos utilizando o método *setCallback* herdado da clase *AsyncClbkContainer* (liña 2136). A execución do servicio asíncrono impleméntase no método *OnEvent* do servidor (liñas 2181-2185) e, unha vez que remata, a notificación ao cliente faise mediante o método *callCallback*, que tamén é herdado da clase *AsyncClbkContainer* (liña 2141).

7.1.2.5. Exemplo da utilización de procesos

A Figura 7.7.a mostra un exemplo de aplicación formada por un proceso de tipo A conectado cun de tipo B e dous de tipo C. Nas conexións entre o proceso tipo A e os procesos tipo C intercambiase un tipo de mensaxes e nas conexións entre o proceso tipo A e o proceso tipo B intercambiase outro tipo diferente. A Figura 7.7.b mostra o diagrama de obxectos da aplicación no que se utilizan as clases explicadas anteriormente, e que é implementada mediante o código seguinte:

```
2240. // Mensaxes
2241. struct VMMessageA : public IVMMsg {};
2242. struct VMMessageB : public IVMMsg {};
2243.
2244. // Portas de enlace
2245. typedef Port<VMMessageA> VMPortA;
2246. typedef Port<VMMessageB> VMPortB;
2247.
2248. // Canais de comunicación
2249. typedef Channel<VMMessageA> VMChannelA;
2250. typedef Channel<VMMessageB> VMChannelB;
2251.
2252. // Proceso tipo A
2253. class VMProcessA : public VMProcess
2254. {
2255. public:
2256.   VMProcessA();
2257.   ~VMProcessA();
2258. };
2259.
2260. VMProcessA::VMProcessA()
2261. {
2262.   VMPortA *port_a1 = new VMPortA("a1");
2263.   VMPortA *port_a2 = new VMPortA("a2");
2264.   VMPortB *port_b1 = new VMPortB("b1");
2265.   VMPortB *port_b2 = new VMPortB("b2");
2266.   ports.insert(port_a1);
2267.   ports.insert(port_a2);
2268.   ports.insert(port_b1);
2269.   ports.insert(port_b2);
2270. }
2271.
2272. VMProcessA::~VMProcessA()
2273. {
2274.   ports.destroy();
2275. }
2276.
2277. // Proceso tipo B
2278. class VMProcessB : public VMProcess
2279. {
2280. public:
2281.   VMProcessB();
2282.   ~VMProcessB();
2283. };
2284.
2285. VMProcessB::VMProcessB()
2286. {
2287.   VMPortB *port_b1 = new VMPortB("b1");
2288.   VMPortB *port_b2 = new VMPortB("b2");
2289.   ports.insert(port_b1);
2290.   ports.insert(port_b2);
2291. }
```

```
2292. VMProcessB::~VMProcessB()
2293. {
2294.  ports.destroy();
2295. }
2296.
2297. // Proceso tipo C
2298. class VMProcessC : public VMProcess
2299. {
2300. public:
2301.  VMProcessC();
2302.  ~VMProcessC();
2303. };
2304.
2305. VMProcessC::VMProcessC()
2306. {
2307.  VMPortA *port_a1 = new VMPortA("a1");
2308.  ports.insert(port_a1);
2309. }
2310.
2311. VMProcessC::~VMProcessC()
2312. {
2313.  ports.destroy();
2314. }
2315.
2316. // Creación e conexión dos procesos
2317. int main()
2318. {
2319.  // canais
2320.  VMChannelA channel_a1, channel_a2;
2321.  VMChannelB channel_b1, channel_b2;
2322.  // procesos
2323.  VMProcessA process_a;
2324.  VMProcessB process_b;
2325.  VMProcessC process_c1, process_c2;
2326.  // conexión A-C1
2327.  process_a.connect("a1", &channel_a1);
2328.  process_c1.connect("a1", &channel_a1);
2329.  // conexión A-C2
2330.  process_a.connect("a2", &channel_a2);
2331.  process_c2.connect("a1", &channel_a1);
2332.  // conexións A-B
2333.  process_a.connect("b1", &channel_b1);
2334.  process_b.connect("b1", &channel_b1);
2335.  process_a.connect("b2", &channel_b2);
2336.  process_b.connect("b2", &channel_b2);
2337.  return 0;
2338. }
```

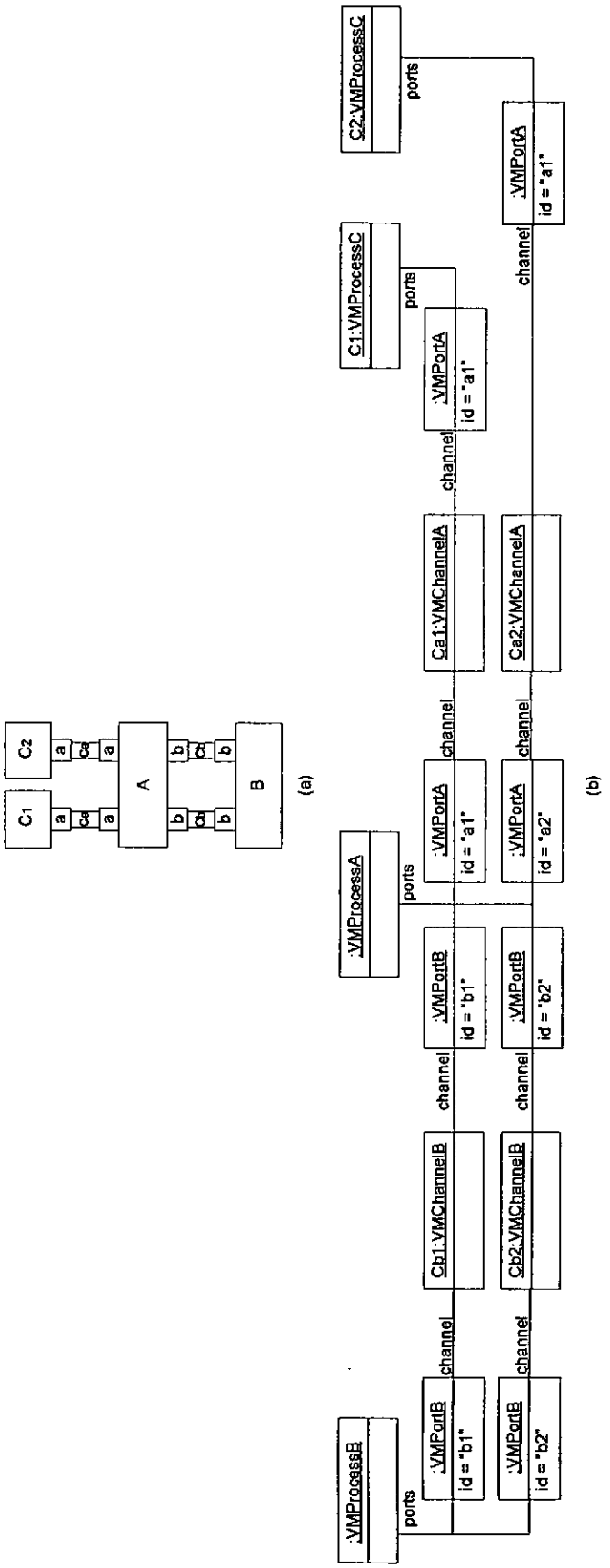


Figura 7.7. Exemplo de utilización dos procesos: a) arquitectura da aplicación; e b) diagrama de obxectos da implementación.

7.1.3. A arquitectura de procesos da máquina virtual

Os procesos que forman a arquitectura lóxica da máquina virtual son mostrados na Figura 7.8. Os cadros pequenos representan as portas de enlace dos procesos, e as frechas que os unen, os canais de comunicación a través dos que se intercambian mensaxes asíncronas (§7.1.2.4.1). Os rectángulos cun contorno máis fino correspóndense con procesos que implementan a interface *IModule* (§7.1.1.1), polo que poden ser substituídos dinamicamente sen deter a execución da máquina virtual. Nesta arquitectura poden distinguirse catro subsistemas principais:

1. *O núcleo da máquina virtual*, formado pola base de datos de E/S na que se almacenan os valores das variábeis de proceso, o manexador de temporizacións, e o depurador.
2. *O subsistema de E/S*, formado polos "drivers" de E/S e os procesos que xestionan a relación entre estes e o núcleo da máquina virtual.
3. *O xestor da configuración*, que proporciona unha interface de acceso remoto aos servicios de configuración e control da máquina virtual.
4. *O subsistema de aplicación*, no que se executan as aplicacións desenvolvidas polo usuario que son cargadas dinamicamente na memoria da máquina virtual.

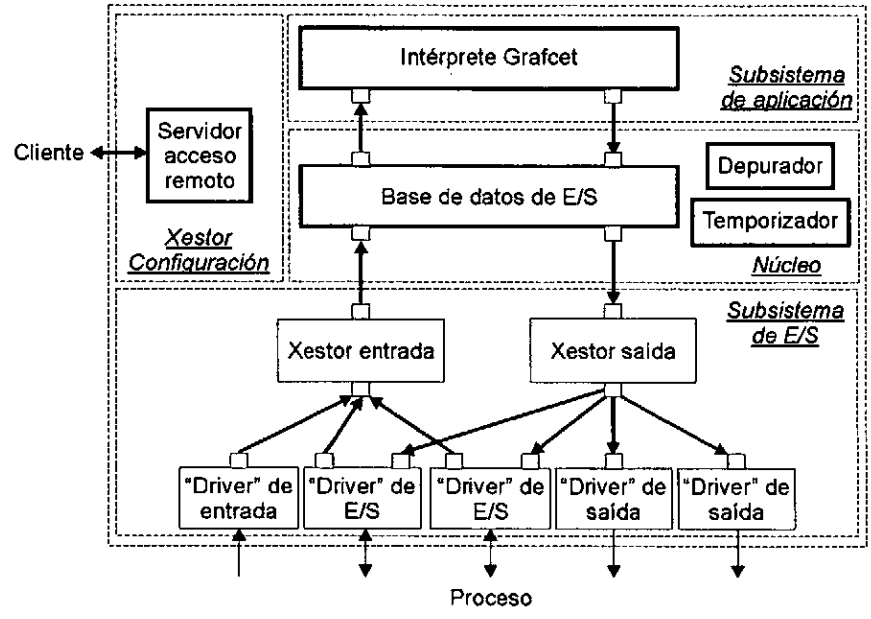


Figura 7.8. Arquitectura de procesos da máquina virtual.

7.1.4. Implementación da arquitectura da máquina virtual

Os diagramas da Figura 7.9 e da Figura 7.10 mostran as clases e relacións utilizadas para representar a estructura da máquina virtual, que é modelada mediante a clase *VMachine*. As instancias desta clase son procesos que inclúen os mecanismos para o almacenamento de módulos (§7.1.1.3) e a xestión de configuracións (§7.1.1.4) derivándoos das clases *ModuleStore* e *ConfigurationStore* respectivamente. A súa estructura se forma mediante a agregación de módulos (§7.1.1.2), utilizando o soporte derivado da clase *StructuredModule*.

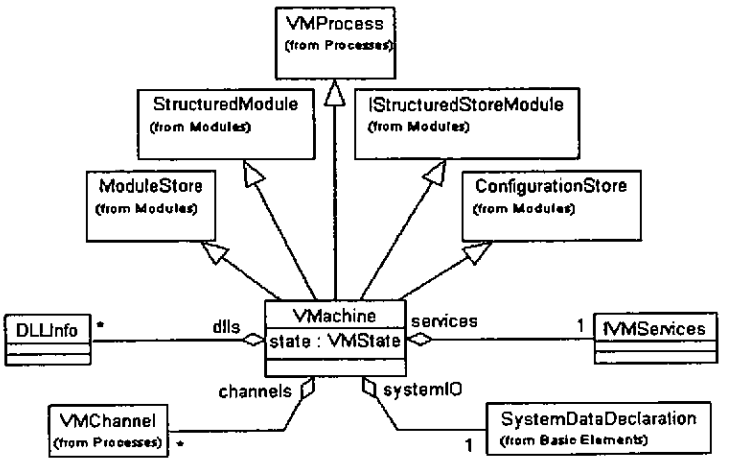


Figura 7.9. Diagrama das clases que modelan a arquitectura da máquina virtual.

As agregacións representadas na Figura 7.9 son as seguintes:

1. *dlls,* almacena a información sobre as DLLs e módulos cargados na memoria da máquina virtual.
2. *systemIO,* almacena a declaración das variábeis de proceso e a súa asignación aos puntos de E/S (§7.2.1.4.1).
3. *channels,* almacena os canais utilizados para o paso de mensaxes entre os procesos da máquina virtual (§7.1.2.4.1).
4. *services*, mantén unha referencia a unha instancia dunha clase derivada da interface *IVMServices*, que proporciona un punto de acceso común aos servicios proporcionados polo núcleo da máquina virtual (§7.4.3).

As clases e relacións utilizadas para a representación dos subsistemas da máquina virtual móstranse na Figura 7.10. As agregacións *output_mgr*, *input_mgr* e *iodrivers* referencian xestores e "drivers" que forman o subsistema de E/S. O núcleo da máquina virtual está formado polas agregacións *iortdb*, *timer* e *debugger*, que referencian respectivamente á base de datos de E/S (§7.3.1), ao xestor de temporizacións (§7.3.2) e ao depurador. O subsistema de xestión da configuración está formado pola agregación *comm_server*, que referencia ao servidor de acceso remoto, e as agregacións *dlls* e *systemIO* (Figura 7.9) comentadas anteriormente. Por último, o subsistema de aplicación está formado unicamente por un intérprete Grafcet referenciado mediante a agregación *player*.

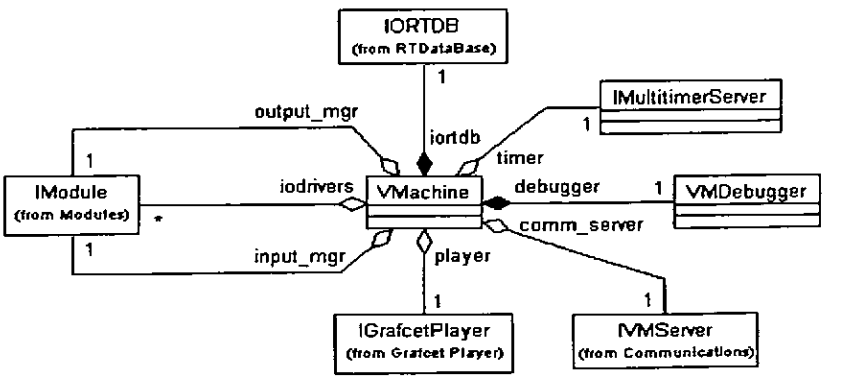


Figura 7.10. Diagrama das clases que modelan os subsistemas da máquina virtual.

Nótese que a representación do subsistema de aplicación mediante a clase abstracta *IGrafcetPlayer* implica que a versión actual da máquina virtual non permite a integración de calquera tipo de interprete ou aplicación. Será preciso modificar este aspecto en futuras versións para permitir a reutilización da arquitectura.

7.2. O subsistema de E/S

O subsistema de E/S (Figura 7.8) está formado polos "drivers" de E/S e os procesos que xestionan a interacción entre os "drivers" e o núcleo da máquina virtual. Nesta sección se explican os detalles do deseño e implementación dos "drivers" (§7.2.1) e xestores de E/S (§7.2.2), así como as técnicas utilizadas para a simulación de valores do proceso (§7.2.3).

7.2.1. Os "drivers" de E/S

A interacción da máquina virtual co proceso realízase mediante dispositivos que proporcionan acceso as súas magnitudes físicas. Estes dispositivos poden ser de moi diferentes tipos: tarxetas de E/S que proporcionen un número reducido de valores booleanos e analóxicos, sistemas de E/S distribuídos, buses de campo, redes de comunicación, etc. O manexo desta variedade de dispositivos realízase mediante "drivers". Cada "driver" da máquina virtual pode xestionar un ou varios dispositivos externos, ben accedendo directamente ao seu "hardware" ou interactuando cos "drivers" propietarios proporcionados polo fabricante do dispositivo. A complexidade da implementación dun "driver" da máquina virtual dependerá do dispositivo concreto e das funcionalidades que este proporcione, pero o seu manexo dende a máquina virtual será igual en cada caso. Ademais os "drivers" implementan a interface *IModule* (§7.1.1.1) polo que poden ser cargados e descargados dinamicamente en memoria dende unha DLL. En consecuencia, a inclusión do soporte a novos dispositivos require da implementación de novos "drivers" ou a modificación dos existentes, pero non da modificación da arquitectura lóxica da máquina virtual.

7.2.1.1. As funcionalidades dun "driver" de E/S

Os "drivers" da máquina virtual proporcionan as seguintes funcionalidades:

1. O control do estado de execución do "driver".
2. A consulta e modificación da configuración do dispositivo de E/S.
3. A lectura e escritura de valores no dispositivo de E/S.
4. A xestión da asignación de variábeis a magnitudes físicas accedidas mediante o dispositivo de E/S.

O seguinte código mostra a declaración da interface *IDeviceDriver* que define os métodos que proporcionan as funcionalidades indicadas:

```
2339.class IDeviceDriver
2340.{
2341.public:
2342.  // control do estado de execución
2343.  virtual void Start() = 0;
2344.  virtual void Suspend() = 0;
2345.  virtual void Resume() = 0;
2346.  virtual void Finish() = 0;
2347.  virtual bool isSuspended() = 0;
2348.  virtual bool isRunning() = 0;
```

```
2349.  // configuración do dispositivo
2350.protected:
2351.  virtual DeviceInfo* buildDeviceInfo(void) const = 0;
2352.  virtual void configureDevice(const DeviceInfo& di) = 0;
2353.public:
2354.  virtual bool getDeviceInfo(DeviceInfo& di) const = 0;
2355.  virtual bool setDeviceInfo(const DeviceInfo& di) = 0;
2356.  virtual bool isValidCharacteristic(const string& c) const = 0;
2357.  virtual bool isValidIOPoint(const string& iopt) const = 0;
2358.  // asignación de variábeis
2359.  virtual bool scheduleOutput(const string& iopt, const string& out) = 0;
2360.  virtual bool unscheduleOutput(const string& iopt) = 0;
2361.  virtual bool scheduleInput(const string& iopt, const string& in,
2362.                             unsigned sz, long period = 0) = 0;
2363.  virtual bool unscheduleInput(const string& iopt) = 0;
2364.  virtual bool unscheduleAll(void) = 0;
2365.  // lectura e escritura de valores
2366.  virtual bool readIOPoint(const string& iopt, void* buf, unsigned sz) = 0;
2367.  virtual bool writeIOPoint(const string& iopt, void* buf, unsigned sz) = 0;
2368.  virtual bool listenIOPoint(const string& iopt, void* buf, unsigned sz,
2369.                             pfn clbk, IAsyncClbkArg* clbk_arg = NULL) = 0;
2370.  virtual bool receivedIOPoint(const string& iopt, unsigned& sz, unsigned& err) = 0;
2371.  virtual bool stopListeningIOPoint(const string& iopt) = 0;
2372.};
```

A interface declara métodos comúns aos da interface *IVMProcess* (§7.1.2.1) para o control e consulta do estado de execución do "driver" (liñas 2343-2348). A configuración do dispositivo realízase mediante os métodos declarados nas liñas 2351-2352. Nótese que os métodos *buildDeviceInfo* e *configureDeviceInfo* están declarados como *protected*, polo que unicamente poden ser utilizados na implementación das clases derivadas. Os métodos para a xestión da asignación de variábeis a magnitudes físicas decláranse nas liñas 2359-2364, e os que permiten ler e escribir valores no dispositivo nas liñas 2366-2371.

Nos apartados seguintes explícanse os detalles da implementación das funcionalidades proporcionadas polos "drivers" da máquina virtual.

7.2.1.2. Configuración dun "driver" de E/S

Cada "driver" da máquina virtual almacena internamente a información que describe a estructura, características e valores que permiten configurar o funcionamento dun dispositivo de E/S. Esta información é creada durante a iniciación do "driver" na implementación do método *buildDeviceInfo* (liña 2351), e pode ser consultada e modificada utilizando os métodos *getDeviceInfo* (liña 2354) e *SetDeviceInfo* (liña 2355). A configuración do dispositivo utilizando os valores da configuración actual é realizada na implementación do método *configureDeviceInfo* (liña 2352).

Cada dispositivo de E/S é modelado na máquina virtual como un conxunto de subsistemas que forman unha estructura xerárquica en forma de árbore. A raíz da xerarquía correspóndense co dispositivo e os niveis inferiores (as follas) cos transductores (denominados *puntos de E/S* na máquina virtual) que permiten ler e escribir os valores das magnitudes físicas do proceso. Cada subsistema ten asociado un conxunto de características que permiten configuralo. Cada característica ten un nome, un conxunto de valores válidos e un valor actual. Os subsistemas, características e valores válidos poden ter asociada unha condición que permite determinar o seu estado de activación ou validez en función dos valores actuais da configuración.

Esta especificación é suficientemente xenérica como para permitir a descrición de múltiples tipos de dispositivos como, por exemplo, a porta serie dun PC, que pode ser modelada como un único punto de E/S cun pequeno conxunto de características (velocidade, bits de parada, tipo de

paridade, etc.); unha tarxeta de adquisición de datos con varios subsistemas (convertedor D/A, convertedor A/D, módulos de E/S dixitais e analóxicas, contadores, temporizadores, etc.) e múltiples puntos de E/S; ou dispositivos de E/S distribuída máis complexos.

Ademais, a posibilidade de asociar unha condición de activación ou validez aos subsistemas, características e valores válidos permite modelar situacións nas que os valores ou a validez dunha característica dependen dos valores doutras. Por exemplo, nunha porta serie o valor do tipo de paridade non é aplicábel se non está activado o control de paridade. Do mesmo xeito poden modelarse diferentes configuracións dos subsistemas que formen o dispositivo de E/S. Por exemplo, hai tarxetas de adquisición de datos analóxicos que teñen dous modos de funcionamento, por voltaxe simple ou por diferencia de voltaxe. No segundo modo, para cada valor adquirido utilízanse dous canais de entrada (o valor adquirido é a diferencia das medicións nambos canais) polo que o número de puntos de E/S é a metade dos dispoñíbeis cando se utiliza o modo simple.

No diagrama da Figura 7.11 móstranse as clases utilizadas para representar a información de configuración dun "driver" da máquina virtual. No resto deste apartado descríbese en detalle cada clase e mostrase un exemplo de como son utilizadas para modelar un dispositivo de E/S, en concreto a porta serie dun PC.

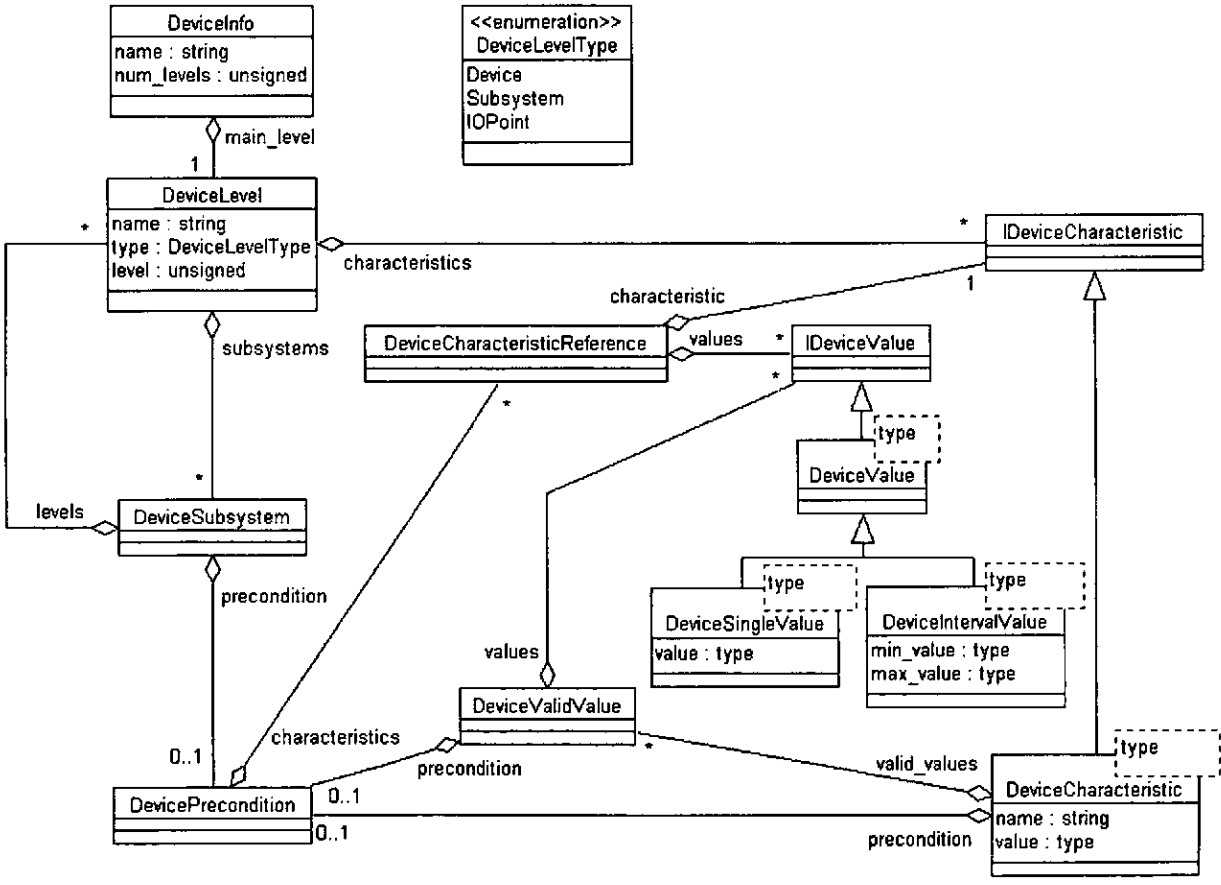


Figura 7.11. Diagrama das clases que modelan a configuración dun dispositivo de E/S.

Tipo de subsistema ("DeviceLevelType")

Enumeración que describe os tipos de subsistemas dun dispositivo de E/S. Os valores escalares definidos son:

- *device*, identifica ao dispositivo. En cada configuración unicamente pode haber un subsistema deste tipo, que se corresponde coa raíz da estructura xerárquica que forman os subsistemas do dispositivo.
- *subsystem*, indica un subsistema xenérico que pode conter outros subsistemas ou puntos de E/S.
- *iopoint*, identifica un transductor de entrada ou saída do dispositivo. Os subsistemas deste tipo unicamente poden aparecer no nivel máis baixo da estructura xerárquica e cada un deles permitirá a lectura ou escritura do valor dunha magnitude do proceso.

Información do dispositivo ("DeviceInfo")

Esta clase é a que agrupa a información do dispositivo e sirve como punto de acceso á estructura xerárquica que esta forma.

Atributos:

- *name*, identificador alfanumérico do dispositivo de E/S.
- *num_levels*, número de niveis da estructura xerárquica que forman os subsistemas do dispositivo.

Asociacións:

- *main_level*, referencia á raíz da estructura xerárquica.

Nivel da xerarquía de subsistemas ("DeviceLevel")

Esta clase representa os niveis da estructura xerárquica que forman os subsistemas.

Atributos:

- *name*, identificador alfanumérico do nivel.
- *type*, tipo de subsistema que representa o nivel.
- *level*, profundidade do nivel na xerarquía.

Asociacións:

- *characteristics*, conxunto de características que permiten configurar o nivel.
- *subsystems*, conxunto de subsistemas que forman o nivel.

Subsistema ("DeviceSubsystem")

Esta clase representa os subsistemas do dispositivo.

Asociacións:

- *precondition*, condición de activación do subsistema.
- *levels*, conxunto de referencias aos niveis inmediatamente inferiores da xerarquía que forman parte do subsistema.

Característica ("IDeviceCharacteristic")

Interface abstracta da que derivan todas as clases utilizadas para representar as características do dispositivos de E/S.

Característica ("DeviceCharacteristic")

Clase parametrizada derivada de *IDeviceCharacteristic*. As clases que representan características asociadas a valores de diferentes tipos (booleanos, enteiros, reais, etc.) obtéñense instanciando esta clase utilizando o tipo de valor como argumento.

Atributos:

- *name*, identificador alfanumérico da característica.
- *value*, valor actual da característica.

Asociacións:

- *precondition*, condición de validez da característica.
- *valid_values*, conxunto de valores válidos da característica.

Valor válido ("DeviceValidValue")

Esta clase representa un dos valores válidos dunha característica. Cada valor válido ten asociada unha condición que indica cando é aplicábel e un conxunto de valores discretos que poden ser asignados á característica.

Asociacións:

- *precondition*, condición de aplicabilidade do valor.
- *values*, conxunto de valores discretos que poden ser asignados á característica.

Valor ("IDeviceValue")

Interface abstracta da que derivan as clases utilizadas para representar os valores que poden ser asignados a unha característica.

Valor ("DeviceValue")

Clase parametrizada derivada de *IDeviceValue*. As clases que representan valores discretos de diferentes tipos (booleanos, enteiros, reais, etc.) obtéñense instanciando esta clase utilizando o tipo de valor como argumento. Os valores poden ser únicos, representados mediante a clase *DeviceSingleValue*, ou estar contidos nun intervalo de valores válidos, representado mediante a clase *DeviceIntervalValue*.

Condición de validez ("DevicePrecondition")

Esta clase representa unha condición de activación ou validez que pode asociarse a subsistemas, características ou valores válidos. A condición consiste nun conxunto de características e dos valores que estas poden ter asignados para que a condición sexa certa.

Asociacións:

- *characteristics*, conxunto de características cuxos valores afectan ao cumprimento da condición de validez.

Referencia dunha característica ("DeviceCharacteristicReference")

Esta clase representa cada característica comprobada para avaliar o resultado da condición de validez.

Asociacións:

- *characteristic*, referencia á característica cuxos valores afectan ao cumprimento da condición de validez.
- *values*, conxunto de valores que a característica pode ter para que se cumpra a condición.

7.2.1.2.1. Exemplo de configuración dun dispositivo de E/S

O código seguinte mostra unha versión simplificada da implementación do método *buildDeviceInfo* nun "driver" que controle a porta serie dun PC:

```
2373. DeviceInfo* COMDriver::buildDeviceInfo() const
2374. {
2375.   // información do dispositivo
2376.   DeviceLevel* com_port = new DeviceLevel("COM1", IOPOINT, 1);
2377.   DeviceInfo* info = new DeviceInfo(com_port);
2378.
2379.   // velocidade
2380.   DeviceCharacteristic<unsigned>*
2381.     baud_rate = new DeviceCharacteristic<unsigned>("Baud Rate", 9600);
2382.   com_port->characs.insert(baud_rate);
2383.
2384.   // paridade
2385.   DeviceCharacteristic<bool>*
2386.     parity_check = new DeviceCharacteristic<bool>("Parity Check", true);
2387.   com_port->characs.insert(parity_check);
2388.
2389.   // número de bits, 5-8
2390.   DeviceCharacteristic<unsigned>*
2391.     num_bits = new DeviceCharacteristic<unsigned>("Number of bits", 7);
2392.   DeviceValidValue* num_bits_vv1 = new DeviceValidValue();
2393.   DeviceIntervalValue<unsigned>*
2394.     num_bits_val1 = new DeviceIntervalValue<unsigned>(5, 8);
2395.   num_bits_vv1->values.push_back(num_bits_val1);
2396.   num_bits->validValues.push_back(num_bits_vv1);
2397.   com_port->characs.insert(num_bits);
2398.
2399.   // bits de parada: 0,1,2 = 1, 1.5, 2
2400.   DeviceCharacteristic<unsigned>*
2401.     stop_bits = new DeviceCharacteristic<unsigned>("Stop bits", ONEBIT);
2402.   DeviceValidValue* stop_bits_vv1 = new DeviceValidValue();
2403.   DeviceSingleValue<unsigned>*
2404.     stop_bits_val1 = new DeviceSingleValue<unsigned>(ONEBIT);
2405.   stop_bits_vv1->values.push_back(stop_bits_val1);
2406.   stop_bits->validValues.push_back(stop_bits_vv1);
2407.
2408.   DeviceValidValue* stop_bits_vv2 = new DeviceValidValue();
2409.   DeviceSingleValue<unsigned>*
2410.     stop_bits_val2 = new DeviceSingleValue<unsigned>(ONE5BITS);
2411.   stop_bits_vv2->values.push_back(stop_bits_val2);
2412.   DevicePrecondition* stop_bits_pre1 = new DevicePrecondition();
2413.   DeviceCharacteristicReference*
2414.     stop_bits_ref1 = new DeviceCharacteristicReference(num_bits);
2415.   DeviceSingleValue<unsigned>*
2416.     num_bits_pval1 = new DeviceSingleValue<unsigned>(5);
2417.   stop_bits_ref1->values.push_back(num_bits_pval1);
2418.   stop_bits_pre1->characteristics.insert(stop_bits_ref1);
2419.   stop_bits_vv2->setPrecondition(stop_bits_pre1);
2420.   stop_bits->validValues.push_back(stop_bits_vv2);
2421.   DeviceValidValue* stop_bits_vv3 = new DeviceValidValue();
2422.   DeviceSingleValue<unsigned>*
2423.     stop_bits_val3 = new DeviceSingleValue<unsigned>(TWOBITS);
2424.   stop_bits_vv3->values.push_back(stop_bits_val3);
2425.   DevicePrecondition* stop_bits_pre2 = new DevicePrecondition();
```

```
2426.  DeviceCharacteristicReference*
2427.     stop_bits_ref2 = new DeviceCharacteristicReference(num_bits);
2428.  DeviceIntervalValue<unsigned>*
2429.     num_bits_pval2 = new DeviceIntervalValue<unsigned>(6, 8);
2430.  stop_bits_ref2->values.push_back(num_bits_pval2);
2431.  stop_bits_pre2->characteristics.insert(stop_bits_ref2);
2432.  stop_bits_vv3->setPrecondition(stop_bits_pre2);
2433.  stop_bits->validValues.push_back(stop_bits_vv3);
2434.
2435.  com_port->characs.insert(stop_bits);
2436.  return info;
2437.}
```

A porta serie é modelada como un único punto de E/S (liña 2376) con catro características que permiten configuralo: a velocidade de transmisión (liñas 2380-2382), o número de bits por palabra (liñas 2385-2387), o control de paridade (liñas 2390-2397), e o número de bits de parada utilizados (liñas 2400-2433). O número de bits por palabra pode ser un valor entre 5 e 8, o cal é definido mediante un intervalo de valores válidos (liñas 2393-2396). Encanto ao número de bits de parada utilizados poden ser 1, 1.5 ou 2. O valor 1.5 só pode seleccionarse se o número de bits da palabra é 5 e o valor 2 só se o número de bits é diferente a 5. Ambas condicións están representadas mediante condicións de validez nas liñas 2412-2419 e liñas 2425-2432, respectivamente. Na Figura 7.12 móstrase o diagrama de obxectos da información de configuración creada polo código anterior.

7.2.1.3. Lectura e escritura de valores

Os "drivers" da máquina virtual permiten ler e escribir valores directamente nos puntos de E/S dun dispositivo mediante os métodos *readIOPoint* e *writeIOPoint* (liñas 2366 e 2367). Ambos métodos reciben como argumento o identificador alfanumérico do punto de E/S, o "buffer" que contén a información a escribir ou no que se almacenará a información lida, e o tamaño desta información. Nótese que estes métodos non impoñen restricciónss no tipo de datos lidos ou escritos, será a base de datos da máquina virtual a que realice as conversións precisas, como se explica en (§7.3.1).

Os puntos de E/S son identificados mediante a súa rota de acceso ("path") na estructura xerárquica da información de configuración do dispositivo, que se forma de xeito semellante á rota de acceso dun arquivo na árbore de directorios dun sistema operativo. Por exemplo, a rota '*\\dd_b8i8o\iodevice\binputs\binput1*' identifica un punto de E/S denominado *binput1* contido no subsistema *binputs* do dispositivo *iodevice* descrito mediante a información de configuración cuxo identificador é *dd_b8i8o*. Esta rota podería utilizarse para indicar unha entrada dun dispositivo que proporcione oito entradas e oito saídas booleanas, por exemplo.

Ademais dos métodos anteriores, os "drivers" da máquina virtual proporcionan outros tres métodos —*listenIOPoint* (liña 2368), *receivedIOPoint* (liña 2370) e *stopListeningIOPoint* (liña 2371)— que permiten realizar lecturas asíncronas nun punto de E/S. O método *listenIOPoint* é a versión asíncrona do método *readIOPoint*, este método notifica a lectura dun valor ao proceso que iniciou a operación utilizando o mecanismo de notificación asíncrona (§7.1.2.4.2). Os outros dous métodos serven, respectivamente, para consultar o estado e deter a lectura asíncrona dun valor.

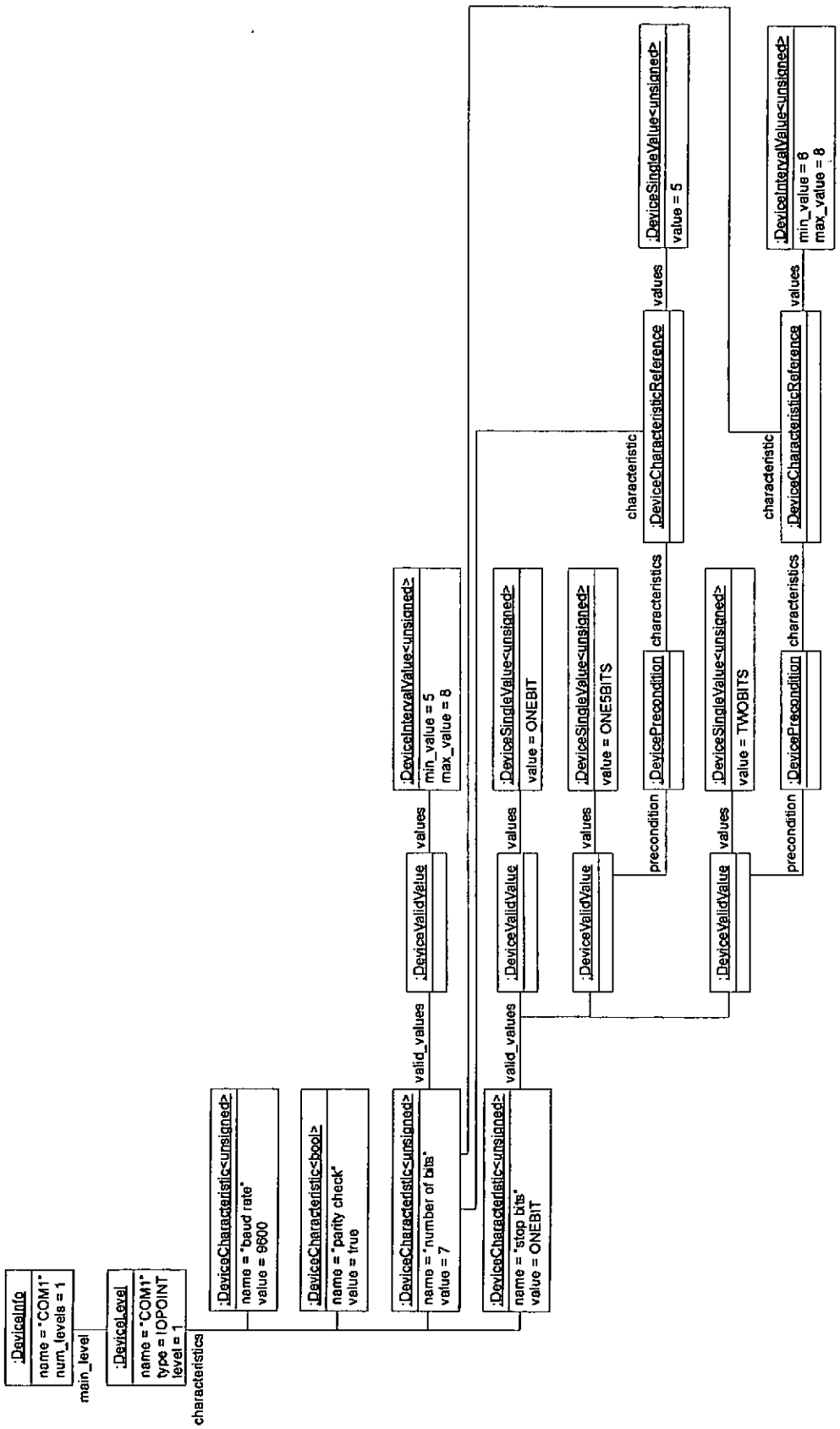


Figura 7.12. Diagrama de obxectos da información de configuración dunha porta serie.

7.2.1.4. Asignación de variábeis a puntos de E/S

Ademais da lectura e escritura de valores simples, os "drivers" da máquina virtual permiten asignar as variábeis definidas nos modelos executados na máquina virtual a puntos de E/S nos dispositivos. Esta asignación permite facer un seguimento continuo das variacións nas entradas dun proceso (indicando como parámetro a frecuencia da monitorización desexada) ou a actualización automática das saídas. Unha vez realizadas as asignacións e iniciados os dispositivos de E/S, o intercambio de valores entre a máquina virtual e os "drivers" faise utilizando o mecanismo de paso de mensaxes (§7.1.2.4.1). Isto permite 'desacoplar' o funcionamento interno da máquina virtual dos aspectos relacionados coa implementación da lectura e escritura de valores nos dispositivos.

Cada "driver" xera mensaxes para as variábeis de entrada que lle foron asignadas ao ritmo indicado pola frecuencia de monitorización. Estas mensaxes, que conteñen os valores actuais das variábeis, son enviadas á máquina virtual. Do mesmo xeito a máquina virtual envía aos "drivers" unha mensaxe para cada variábel de saída que sexa preciso actualizar. A velocidade á que os cambios nos valores das entradas son procesados e os valores das saídas producidos depende do tempo de resposta da máquina virtual, é dicir, o tempo que esta precisa para calcular os valores das saídas a partir das entradas dacordo ao especificado nos modelos executados no subsistema de aplicación. Tal e como se explica no Capítulo 8, este tempo é variábel, e en caso de ser superior ao ritmo de cambio das magnitudes do proceso pode producirse un desbordamento nos canais da máquina virtual debido a que esta non pode procesar todas as mensaxes xeradas polos "drivers". A posibilidade de especificar unha frecuencia de monitorización na asignación de variábeis de entrada a dispositivos de E/S permite reducir en certa medida este problema. O valor dos tempos de monitorización dependerá de cada proceso concreto e será responsabilidade do deseñador do sistema axustalos para evitar o colapso da máquina virtual.

A asignación de variábeis a puntos de E/S nos dispositivos realízase mediante os métodos *scheduleOutput* (liña 2359) e *scheduleInput* (liña 2361). Ambos métodos reciben como argumento o identificador alfanumérico do punto de E/S e o identificador da variábel asignada. No método *scheduleInput,* que asigna unha variábel de entrada, pásase ademais como argumento o tamaño do valor a monitorizar e a frecuencia de actualización na máquina virtual (unha frecuencia igual a cero indica que o valor debe actualizarse cada vez que cambie). As asignacións individuais a un punto de E/S elimínanse cos métodos *unscheduleOutput* (liña 2360) e *unscheduleInput* (liña 2363), que reciben como argumento o identificador alfanumérico do punto de E/S. Tamén poden eliminarse todas as asignacións dun dispositivo utilizando o método *unscheduleAll* (liña 2364).

A clase *DeviceDriver*, derivada de *IDeviceDriver*, proporciona unha implementación por defecto dos métodos anteriores e do mecanismo que xestiona a monitorización das variábeis de entrada e a actualización das de saída. Ademais esta clase tamén define dúas portas de enlace que permiten establecer dúas conexións entre cada "driver" e a máquina virtual, unha para o envío e outra para a recepción de mensaxes. No resto deste apartado explícanse os detalles desta implementación.

7.2.1.4.1. Asignación de variábeis a puntos de E/S

A clase *DeviceDriver* mantén información sobre as variábeis asignadas aos puntos de E/S do dispositivo. A implementación dos métodos *scheduleOutput* e *scheduleInput* controla que unha mesma variábel non é asignada a máis dun punto de E/S, que como moito se asigna unha

variábel a cada punto de E/S e que o punto de E/S ao que se asigna a variábel existe no dispositivo.

7.2.1.4.2. Monitorización de variábeis de entrada

O mecanismo de monitorización de variábeis de entrada xera unha mensaxe co valor de cada variábel de entrada asignada ao dispositivo ao ritmo indicado pola frecuencia de monitorización da variábel. Estas mensaxes son enviadas á máquina virtual a través dunha das dúas portas de enlace definidas por defecto para cada "driver". En función da frecuencia de monitorización as variábeis de entrada son divididas en dous grupos:

1. As variábeis cuxo valor é enviado á máquina virtual cada vez que este cambia (a frecuencia de monitorización é igual a cero).
2. As variábeis cuxo valor é enviado á máquina virtual a intervalos regulares.

Ambos casos son tratados de forma diferente na implementación da clase *DeviceDriver*. No referente ás variábeis do primeiro grupo, para cada unha delas iniciase unha lectura asíncrona no punto de E/S no que estea asignada utilizando o método *listenIOPoint* (liña 2368). Cando se recibe unha notificación procedente dalgunha das operacións de lectura asíncrona iniciadas, o "driver" comproba a que variábel corresponde, crea e envía unha mensaxe á máquina virtual contendo o identificador da variábel e o seu valor, e reinicia unha nova operación de lectura asíncrona no punto de E/S no que a variábel estea asignada. Nótese que os detalles da detección dun cambio no valor da variábel e a súa notificación son manexados pola subclase derivada de *DeviceDriver* que implemente o método *listenIOPoint*, e dependerán de cada dispositivo concreto.

Para o manexo das variábeis do segundo grupo utilízase un temporizador (§7.3.2) de xeito que o "driver" reciba notificacións periódicas e determine en cada instante as variábeis cuxos valores é preciso actualizar. Na implementación da clase *DeviceDriver* o temporizador é iniciado de xeito que se reciba unha notificación cada vez que transcorra un tempo igual ao máximo común divisor (m.c.d.) dos períodos de monitorización das variábeis asignadas ao dispositivo. Para cada notificación recibida calcúlase o conxunto de variábeis a actualizar como:

$$\text{update\_vars} = \{\text{var} \mid \text{time} \bmod T_u(\text{var}) = 0\} \tag{7.1}$$

onde:

- $T_u(var)$, período de actualización da variábel.
- *time*, tempo transcorrido dende a iniciación do temporizador. Este tempo calculase como:

$$\text{time} = \text{ticks} * \text{mcd} \tag{7.2}$$

sendo:

- *ticks*, número de notificacións recibidas.
- *mcd*, m.c.d. dos períodos de monitorización das variábeis.

É dicir, as variábeis a actualizar son aquelas cuxo período de monitorización é un divisor do tempo transcorrido dende a iniciación do temporizador.

O seguinte exemplo mostra o cálculo deste conxunto para a serie de notificacións recibidas nun "driver" con catro variábeis de entrada asignadas con períodos de monitorización de 5, 5, 10 e 15 mseg. respectivamente.

```
2438.// Valores iniciais
2439.vars = {(v1, 5), (v2, 5), (v3, 10), (v4, 15)}
2440.ticks = 0
2441.time = 0
2442.mcd = 5
2443.mcm = 30
2444.ticks = 1 // primeira notificación
2445.time = 5
2446.update_vars = {v1, v2}
2447.ticks = 2 // segunda notificación
2448.time = 10
2449.update_vars = {v1, v2, v3}
2450.ticks = 3 // terceira notificación
2451.time = 15
2452.update_vars = {v1, v2, v4}
2453.ticks = 4 // cuarta notificación
2454.time = 20
2455.update_vars = {v1, v2, v3}
2456.ticks = 5 // quinta notificación
2457.time = 25
2458.update_vars = {v1, v2}
2459.ticks = 6 // sexta notificación
2460.time = 30
2461.update_vars = {v1, v2, v3, v4}
```

Nótese que a serie anterior repítese cun período igual ao mínimo común múltiplo dos períodos de monitorización das variábeis (*mcm* = 30), polo que cando o número de notificacións é tal que o tempo transcorrido sexa igual a ese período (*ticks* = 6), pode reiniciarse o seu valor e considerar que se volve á situación inicial. Na implementación do algoritmo de cálculo do conxunto *update_vars* reduciuse o número de operacións necesarias ordenando as variábeis dacordo ao seu período de monitorización e considerando en cada notificación unicamente aquelas que teñen un período (valor de T_u) inferior ao tempo transcorrido dende a iniciación do temporizador (valor de *time*).

Unha vez determinado o conxunto de variábeis a actualizar o "driver" obtén os seus valores dos puntos de E/S nos que están asignadas utilizando o método *readIOPoint* (liña 2366) e envíalle á máquina virtual as mensaxes con estes valores a través da porta de enlace correspondente. A implementación do método *readIOPoint* faise nas subclases derivadas de *DeviceDriver* e dependerá de cada dispositivo concreto. A técnica descrita reduce o "overhead" debido á monitorización de variábeis, xa que:

1. A interacción entre o temporizador e o "driver" redúcese unicamente á iniciación do temporizador e á recepción e procesamento no "driver" das notificacións do temporizador.
2. O cálculo do conxunto de variábeis a actualizar require unicamente unha división e unha comparación por cada variábel cuxo período de monitorización sexa inferior ao tempo transcorrido dende a iniciación do temporizador.

Sen embargo presenta dous inconvenientes:

1. Cada "driver" utiliza un temporizador, que é normalmente un recurso limitado.
2. Os períodos de monitorización das variábeis deben axustarse para que o seu m.c.d. sexa un valor o suficientemente grande para non colapsar o "driver" con máis notificacións das que poda procesar.

7.2.1.4.3. Actualización de variábeis de saída

A actualización dos valores das saídas realízase mediante a lectura sen bloqueo (§7.1.2.4.1) da porta de enlace do "driver" pola que se reciben as mensaxes enviadas pola máquina virtual. Estas mensaxes conteñen o identificador e valor da variábel de saída a actualizar. Cada vez que se recibe unha mensaxe o "driver" comproba en que porta de E/S está asignada a variábel e escribe nela o novo valor utilizando o método *writeIOPoint* (liña 2367). A implementación deste método faise nas subclases derivadas de *DeviceDriver* e dependerá de cada dispositivo concreto.

7.2.2. Xestión da E/S

A interface entre o núcleo da máquina virtual e os "drivers" de E/S é xestionada mediante dous procesos: o xestor de entradas e o de saídas. Como mostra a Figura 7.13 cada xestor mantén conexións cos "drivers" de dispositivo e coa máquina virtual para o intercambio de mensaxes. O xestor de entradas ten dúas portas de enlace, unha a través da que se reciben as mensaxes dos "drivers" de E/S cos valores das variábeis de entrada do proceso e outra pola que se encamiñan estas mensaxes cara á máquina virtual. Pola súa banda o xestor de saídas recibe por unha porta de enlace as mensaxes da máquina virtual cos valores das variábeis de saída a actualizar e encamíñaas ao dispositivo correspondente utilizando unha porta de enlace por cada "driver" de saída.

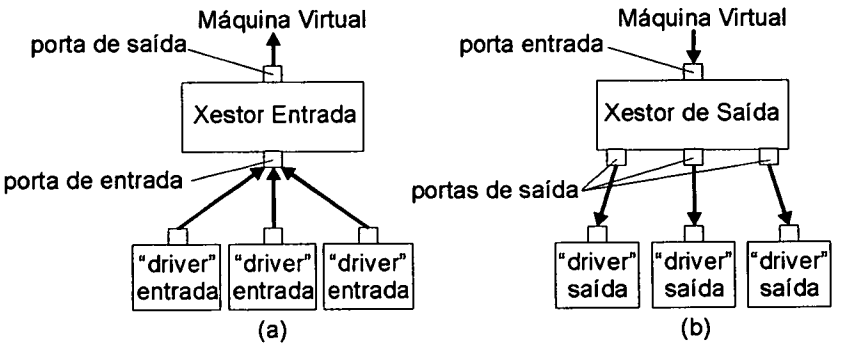


Figura 7.13. Conexións dos xestores de E/S da máquina virtual.

O seguinte código mostra a declaración da interface abstracta *IIOMgr* que implementan os xestores de E/S:

```
2462. struct IIOMgr
2463. {
2464.   // control do estado
2465.   virtual void Start() = 0;
2466.   virtual void Suspend() = 0;
2467.   virtual void Resume() = 0;
2468.   virtual void Finish() = 0;
2469.   virtual bool isSuspended() = 0;
2470.   virtual bool isRunning() = 0;
2471.   // portas de enlace
2472.   virtual bool isValidPort(const string& port, bool& state) = 0;
2473.   virtual bool isConnected(const string& port, bool& state) = 0;
2474.   virtual bool connect(const string& port, VMChannel* channel) = 0;
2475.   virtual bool disconnect(const string& port) = 0;
2476.   virtual bool getConnection(const string& port, VMChannel* channel) = 0;
2477. };
```

Como pode comprobarse esta interface coincide coa interface pública dos procesos da máquina virtual (§7.1.2.1). A Figura 7.14 mostra o diagrama das clases derivadas de *IIOMgr* que son utilizadas para a implementación dos xestores de E/S. A súa implementación por defecto é proporcionada pola clase *IOMgr* utilizando os métodos derivados da clase *VMProcess*. A clase *IOMgr* crea no seu constructor a porta de enlace para a recepción de mensaxes que teñen en común tanto os xestores de entrada como os de saída e implementa o mecanismo de recepción de mensaxes. Ademais a clase *IOMgr* declara un método que é invocado cada vez que unha mensaxe é recibida. Este método é tipo *protected* e será redefinido polas subclases para implementar comportamentos específicos. A súa declaración é a seguinte:

```
2478.class IOMgr : public VMProcess, public IIOMgr
2479.{
2480.protected:
2481.  virtual bool onInputMessage(IVMMsg* msg) = 0;
2482.};
```

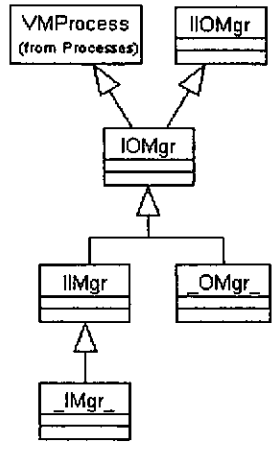


Figura 7.14. Diagrama das clases que modelan os xestores de E/S.

A clase abstracta *IIMgr* é utilizada como clase base dos xestores de entrada e no seu constructor crease a porta de enlace utilizada para a conexión do xestor coa máquina virtual. En canto ás clases *IMgr* e *OMgr*, son as que proporcionan as implementacións por defecto do método *onInputMessage* (liña 2481) para os xestores de entrada e saída, respectivamente. Esta implementación se encarga de encamiñar as mensaxes recibidas á porta de enlace de saída que corresponda, dacordo do tipo de xestor do que se trate.

A clase *OMgr* implementa ademais o manexo das portas de enlace utilizadas para conectar o xestor aos "drivers" de saída. Esta clase redefine os métodos *connect* (liña 2014) e *disconnect* (liña 2015) de xeito que as portas de enlace son creadas e destruídas dinamicamente cada vez que se realiza ou elimina unha conexión entre o xestor e os "drivers" de saída. Cada porta de enlace creada dinamicamente identifícase co nome do "driver" ao que estea conectada. Este identificador utilízase para determinar a qué porta de enlace enviar as mensaxes recibidas polo xestor de saídas (cada mensaxe contén o nome e valor da variábel a actualizar e o identificador do "driver" de saída no que está asignada).

Os xestores implementados polas clases *IMgr* e *OMgr* unicamente proporcionan a funcionalidade de encamiñar as mensaxes intercambiadas entre a máquina virtual e os dispositivos de E/S. Sen embargo derivando novas clases das explicadas anteriormente é posíbel implementar xestores que proporcionen outras funcionalidades adicionais, como por exemplo:

1. O rexistro das mensaxes intercambiadas para a análise posterior do funcionamento do sistema.
2. A utilización de secuencias de mensaxes de entrada preestablecidas (escenarios) para a simulación e a análise da resposta do sistema.
3. O filtrado e a asignación de prioridades ás mensaxes.

Nótese que os xestores de E/S derivan da clase *IModule* (§7.1.1.1), polo que poden ser substituídos en tempo de execución utilizando a técnica explicada en (§7.1.1.5) para cargalos e descargalos dinamicamente en memoria dende unha DLL.

7.2.3. Simulación de E/S

A incorporación dalgún mecanismo co que se poidan simular as entradas do proceso externo e rexistrar os valores de saída producidos como resposta permite utilizar a máquina virtual en diferentes funcións, como por exemplo:

1. A verificación mediante simulación do comportamento do sistema de control antes da súa posta en funcionamento.
2. A utilización conxunta de subprocesos reais e simulados para analizar a repercusión de futuros cambios no sistema.
3. A análise da resposta do sistema ante secuencias de entradas preprogramadas (escenarios) para optimizar a súa resposta.

Na arquitectura proposta para a máquina virtual (Figura 7.8) é posíbel incorporar un mecanismo de simulación de E/S utilizando dúas técnicas diferentes:

1. Substituíndo os xestores de E/S (§7.2.2) básicos por outros que permitan a utilización de entradas simuladas e o rexistro das saídas.
2. Proporcionando "drivers" (§7.2.1) que simulen os dispositivos de E/S.

Tanto os xestores coma os "drivers" de E/S son procesos que implementan a interface *IModule* (§7.1.1.1), polo que poden ser cargados e descargados dinamicamente na memoria da máquina virtual. Mediante a definición das configuracións (§7.1.1.2) axeitadas será posíbel alternar entre o modo de simulación e o de execución sen que sexa preciso deter a máquina virtual. A Figura 7.15 mostra exemplos de configuracións de simulación que utilizan as dúas técnicas indicadas anteriormente. Nótese que ambas técnicas non son excluíntes e poderían combinarse.

No resto deste apartado detállase o mecanismo que se definiu para o intercambio da información de simulación entre os módulos da máquina virtual e os simuladores remotos, explícase a implementación do mecanismo de simulación nun ambiente distribuído sobre redes TCP/IP, e descríbese a implementación dun "driver" xenérico para utilizar en configuracións que apliquen a segunda das técnicas indicadas.

7.2.3.1. O intercambio de información de simulación

No deseño do mecanismo para o intercambio de información entre a máquina virtual e os simuladores remotos tivéronse en conta os seguintes requisitos:

1. Deberá ser posíbel utilizar diferentes simuladores simultaneamente. Os simuladores poderán enviar valores de entrada simulados e/ou recibir valores de saída, e poderán ser

tanto aplicacións específicas como formar parte de ambientes de simulación ou visualización (p.e. LabView, HP VEE, Excel, etc.).

2. Tanto a información intercambiada como o protocolo de comunicación deben ser simples.
3. O intercambio de información debe ser independente do medio de comunicación utilizado, e poderá ser implementado utilizando diferentes protocolos de comunicación (p.e. DDE, OPC, TCP/IP, etc.).

O cumprimento destes requisitos permite a utilización do mecanismo de simulación en configuracións distribuídas heteroxéneas nas que se utilicen diferentes aplicacións de simulación e visualización interactuando mediante diferentes sistemas de comunicación (Figura 7.16).

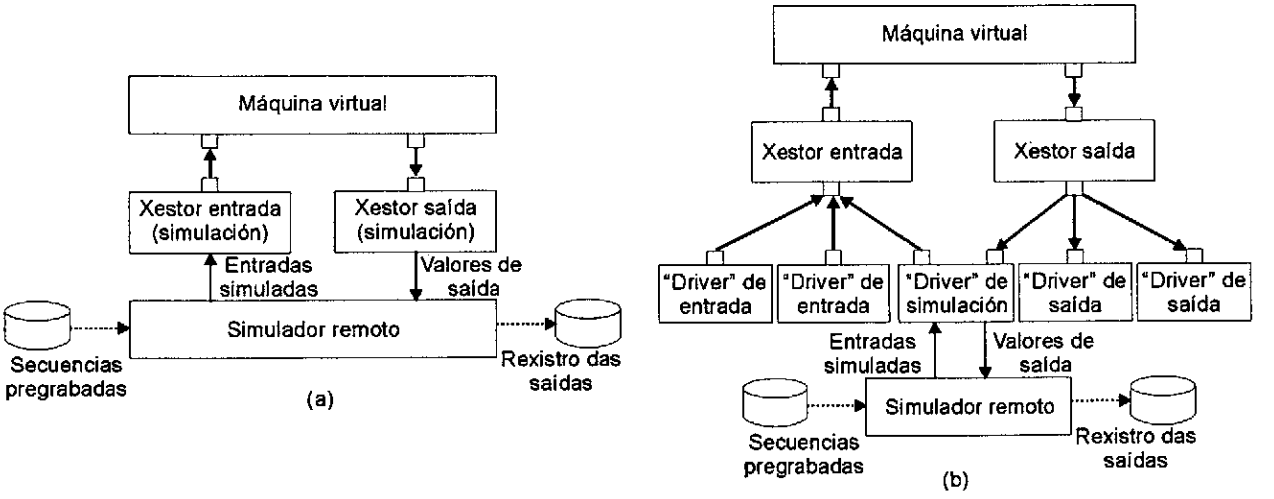


Figura 7.15. Técnicas para a simulación de E/S na máquina virtual: a) utilizando un xestor de E/S; e b) utilizando un "driver".

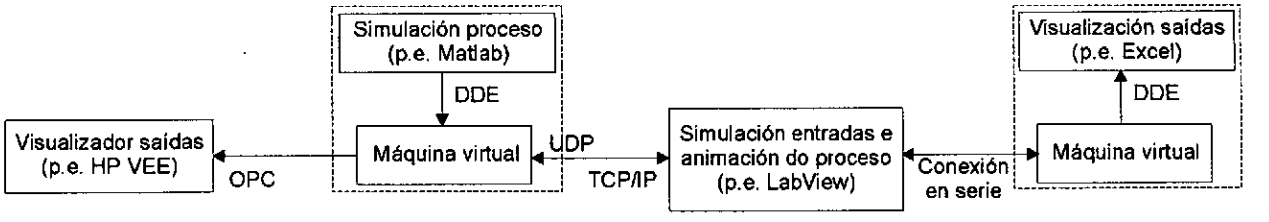


Figura 7.16. Exemplo dun ambiente de simulación distribuído.

7.2.3.1.1. Modelado do intercambio de información de simulación

Os participantes nun ambiente como o mostrado na Figura 7.16 poden clasificarse en dúas categorías non excluíntes dependendo do seu papel no intercambio de información de simulación:

1. *Os productores de valores de entrada* simulados (p.e. simuladores de paneis de operador, simuladores de proceso, etc.).
2. *Os consumidores de valores de saída* (p.e. animacións gráficas do estado do proceso, aplicacións de análise de datos, etc.).

O intercambio de información pode, polo tanto, modelarse como unha interacción entre un conxunto de productores e consumidores de valores de simulación. Esta interacción estará regulada por un mediador [67] que cumprirá dúas funcións:

1. Proporcionar un punto de acceso coñecido no ambiente distribuído ao que os productores e consumidores poden conectarse para enviar e recibir valores.
2. Distribuír cada valor enviado polos productores entre os consumidores que solicitaron recibilo. Para reducir o número de mensaxes intercambiadas os consumidores indicarán, no momento de conectarse ao mediador, as variábeis que queren recibir. Deste xeito o mediador unicamente reenviará cada valor recibido aos consumidores interesados.

A utilización dun mediador permite manexar de maneira transparente diferentes configuracións no intercambio de datos, por exemplo:

1. O mediador proporciona un punto de acceso coñecido único ao que se conecten productores e consumidores utilizando o mesmo protocolo de comunicacións.
2. O mediador proporciona un ou máis puntos de acceso coñecidos tanto aos productores como aos consumidores. Neste caso poden utilizarse protocolos de comunicación diferentes.

Nótese ademais que nun ambiente distribuído poden coexistir múltiples mediadores e que sería tamén posíbel utilizar configuracións nas que un mediador se conecte a outro como productor ou consumidor. Na Figura 7.17 móstrase unha posíbel implementación da configuración da Figura 7.16 utilizando o patrón Productor/Mediador/Consumidor.

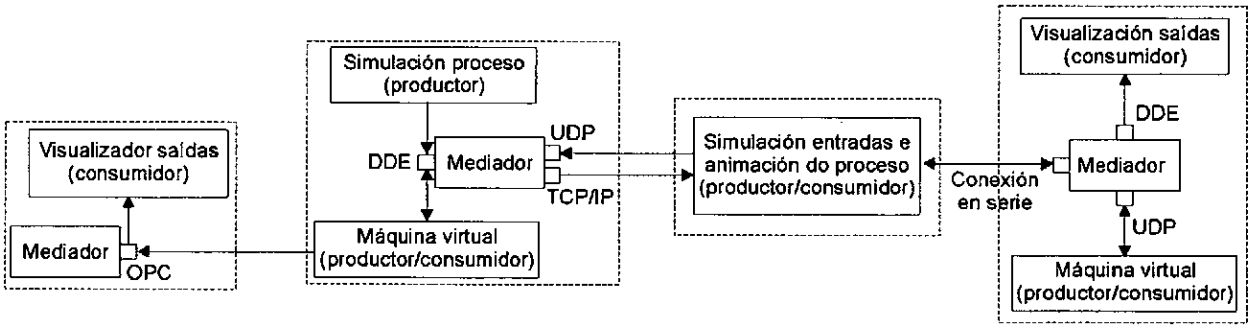


Figura 7.17. Ambiente de simulación distribuído utilizando o patrón Productor / Mediador / Consumidor.

7.2.3.1.2. O formato das mensaxes

A Figura 7.18 mostra a estructura dos dous tipos de mensaxes utilizadas no intercambio de información de simulación. O primeiro tipo de mensaxe é enviado por cada participante no momento de conectarse ao mediador, e está formado polos seguintes campos:

1. Un indicador booleano (1 byte) que indica se o participante quere recibir valores.
2. Un indicador booleano (1 byte) que indica se o participante vai enviar valores.
3. Os nomes das variábeis das que o participante quere recibir valores.

O segundo tipo de mensaxe é o utilizado para intercambiar os valores simulados, e está formado polos seguintes campos:

1. O nome da variábel.
2. O tamaño do valor da variábel (4 bytes).
3. O valor da variábel.

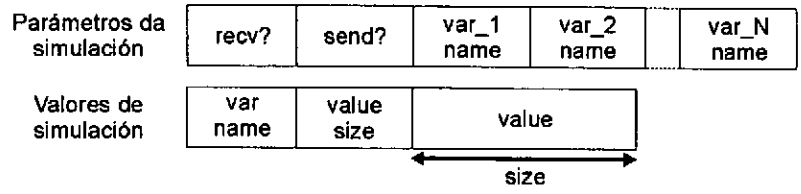


Figura 7.18. Formato das mensaxes utilizadas para o intercambio de información de simulación.

7.2.3.1.3. O protocolo de intercambio de información

A Figura 7.19 mostra un exemplo do protocolo de comunicación utilizado para o intercambio de información de simulación. Cada participante se conecta a un mediador enviando unha mensaxe do primeiro dos tipos explicados no apartado anterior, coa que indica se vai a enviar e/ou recibir valores e que valores quere recibir. Os valores son intercambiados utilizando mensaxes do segundo tipo. Os participantes que simulan valores de entrada envían estes valores ao mediador que se encarga de distribuílas entre os participantes que solicitaron recibir eses valores.

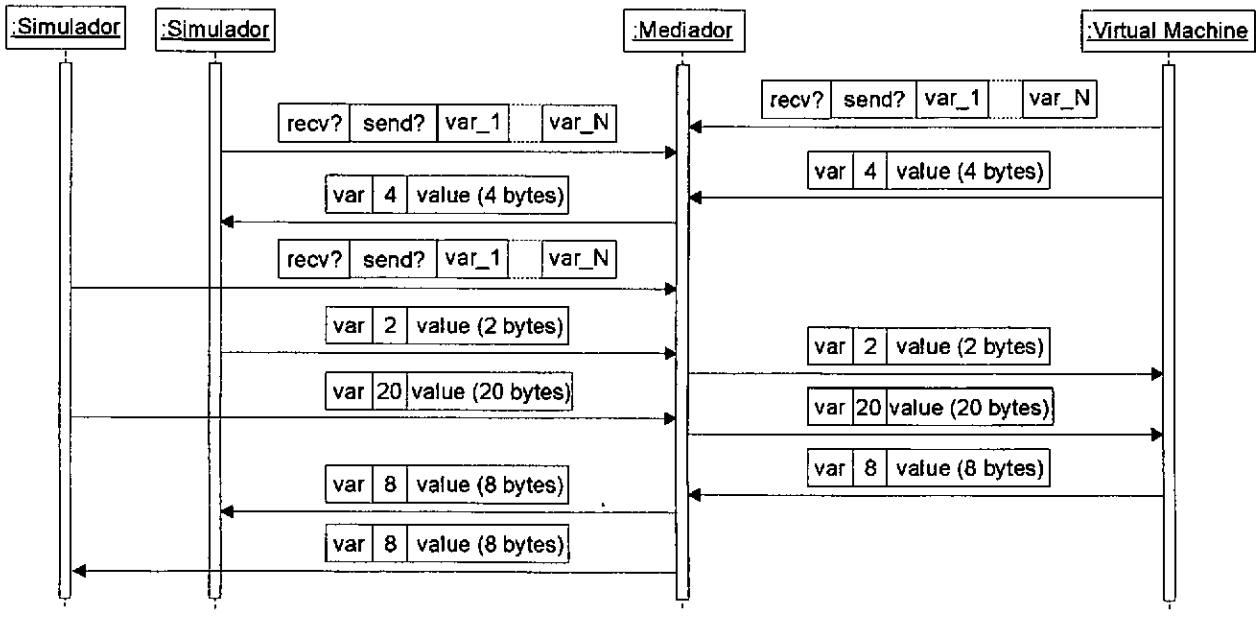


Figura 7.19. Secuencia de mensaxes do protocolo de intercambio de información de simulación.

7.2.3.2. Modelado dun mediador xenérico

A Figura 7.20 mostra o diagrama das clases definidas para representar a estructura dun mediador. A clase abstracta *ISimulationServer* declara a interface común a todos os mediadores utilizados na simulación. O código da declaración desta interface é o seguinte:

```
2483. struct ISimulationServer
2484. {
2485.   virtual void onReceive(ISimulationLinkPoint* link,
2486.                          const SimulationDataSeq& data) = 0;
2487.   virtual bool buildLinkPoints(ISimulationServer* server) = 0;
2488. };
```

Esta interface declara os métodos *onReceive* (liña 2485), que será invocado cada vez que se reciban valores simulados a través dalgunha das conexións abertas do mediador, e *buildLinkPoints* (liña 2487), método utilizado para crear os puntos de acceso do mediador. Os

puntos de acceso son creados invocando este método durante a iniciación do mediador e suponse que non se modifican durante a vida deste. Poderían definirse mediadores que permitiran a construcción e destrucción dinámica dos puntos de acceso mediante a utilización de métodos constructores ("factory methods" [67]) en clases derivadas de *ISimulationServer*.

Figura 7.20. Diagrama das clases utilizadas para modelar un mediador.

Os puntos de acceso do mediador coñecidos polos participantes na simulación son representados mediante a clase abstracta *ISimulationLinkPoint*. O código que declara esta interface é o seguinte:

```
2489.class ISimulationLinkPoint
2490.{
2491.protected:
2492.  virtual void addConnection(ISimulationService* service) = 0;
2493.  virtual void removeConnection(ISimulationService* service) = 0;
2494.public:
2495.  virtual void onReceive(ISimulationService* service) = 0;
2496.  virtual void onDisconnect(ISimulationService* service) = 0;
2497.  virtual void broadcastData(const SimulationDataSeq& data) = 0;
2498.};
```

Os métodos *addConnection* (liña 2492) e *removeConnection* (liña 2493) son utilizados para engadir e eliminar conexións entre o mediador e os participantes; *onReceive* (liña 2495) e *onDisconnect* (liña 2496) son métodos públicos invocados dende as conexións para procesar a recepción de datos e a finalización da conexión, respectivamente; e finalmente, o método *broadcastData* (liña 2497) é invocado dende o mediador para enviar datos a todas as conexións dun punto de acceso.

Por último, a clase abstracta *ISimulationService* modela cada conexión establecida entre o mediador e os participantes. Esta interface declara métodos que serán invocados dende o punto de acceso para o envío e recepción de valores:

```
2499.struct ISimulationService
2500.{
2501.  virtual bool getPendingData(SimulationDataSeq& data) = 0;
2502.  virtual void sendData(const SimulationDataSeq& data) = 0;
2503.};
```

7.2.3.3. Implementación dun mediador en redes TCP/IP

A Figura 7.21 mostra o diagrama das clases utilizadas na implementación dun mediador que permite intercambiar datos de simulación entre unha ou máis máquinas virtuais e diferentes simuladores remotos utilizando o protocolo TCP/IP. Este mediador é un proceso "multithread" que proporciona dous puntos de acceso ("sockets"), nun deles se conectan os simuladores e noutro as máquinas virtuais. O mediador encárgase de enviar os valores de cada variábel recibidos polas conexións dun "socket" ás conexións do outro que solicitaran recibir valores desa variábel.

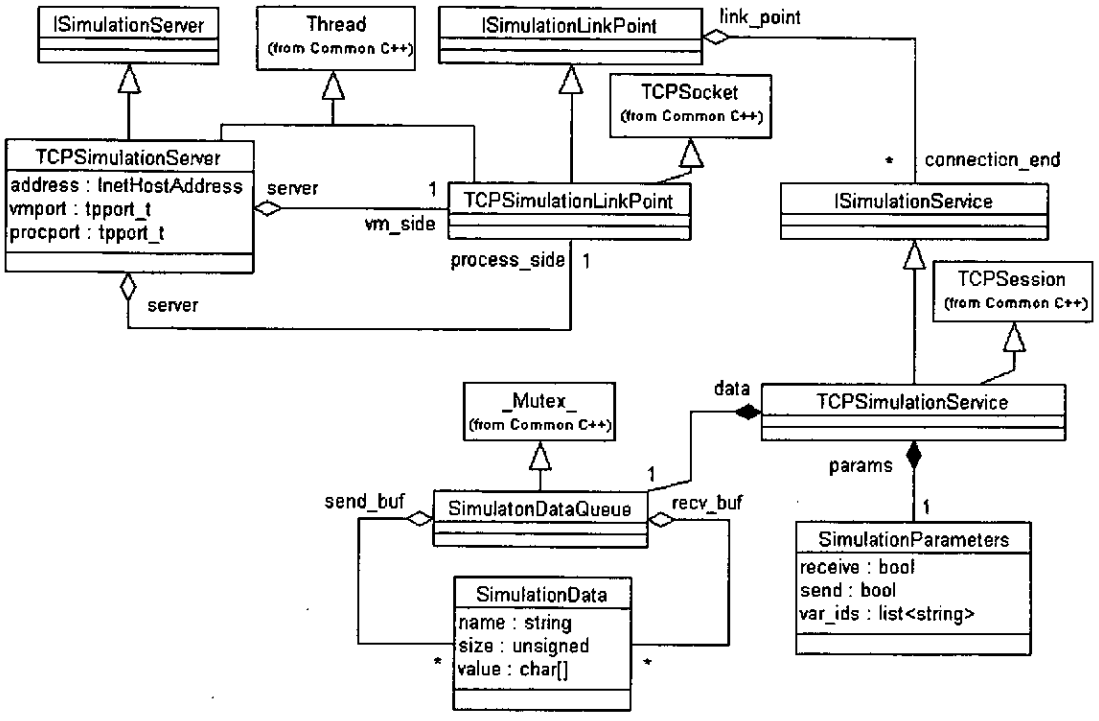


Figura 7.21. Diagrama das clases utilizadas para implementar o mediador TCP/IP.

O mediador é unha instancia da clase *TCPSimulationServer*, que deriva da clase *Thread* da libraría Common C++ (§1.3.2). As instancias desta clase son "threads" que implementan a interface *ISimulationServer* e proporcionan dous puntos de acceso —agregacións *vm_side* e *process_side*— para a conexión dos participantes na simulación. Os puntos de acceso son instancias da clase *TCPSimulationLinkPoint*, que deriva das clases *TCPSocket* e *TCPThread* da libraría Common C++. Cada punto de acceso é, en consecuencia, un servidor TCP que implementa a interface *ISimulationLinkPoint*. As conexións con estes servidores son manexadas por instancias da clase *TCPSimulationService*, que deriva da clase *TCPSession* da libraría Common C++. Cada instancia desta clase é, polo tanto, un "thread" que manexa unha conexión entre dous "sockets". En cada conexión se almacenan os parámetros de configuración enviados polo participante ao conectarse —agregación *params*— e mantéñense dúas colas cos valores recibidos e os pendentes de enviar a través da conexión —agregación *data*—.

A Figura 7.22 mostra a secuencia de mensaxes intercambiados entre un simulador, o mediador e unha máquina virtual para o establecemento de conexións e o intercambio de valores de simulación. Como complemento deste mediador creouse un VI[74] en LabView que permite simular paneis de E/S booleanas e intercambiar os seus valores co mediador TCP/IP.

[74] Os detalles da implementación deste VI non foron incluidos como parte desta documentación.

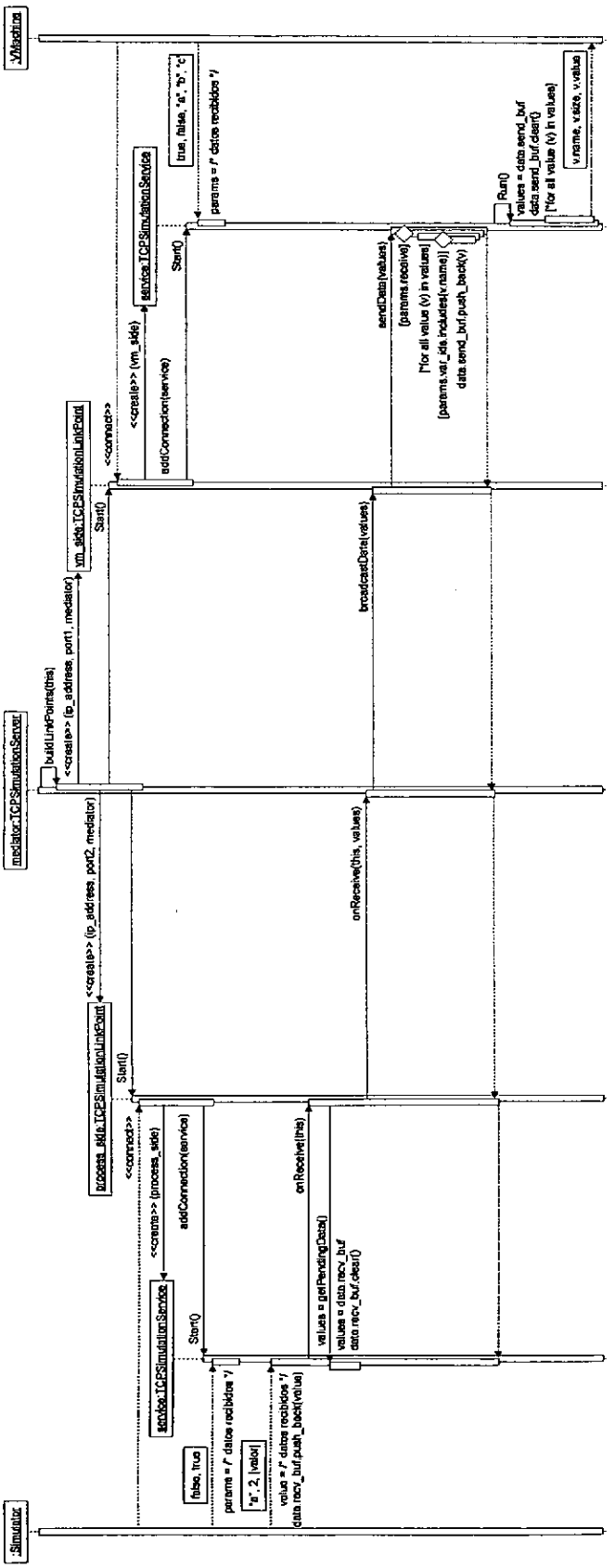


Figura 7.22. Secuencia de mensaxes do intercambio de información de simulación utilizando un mediador TCP/IP.

7.2.3.4. Implementación dun simulador de dispositivos de E/S

Como parte da máquina virtual incluíuse a implementación dun simulador de dispositivos de E/S que intercambia valores a través dunha rede TCP/IP e que pode ser utilizado conxuntamente co mediador explicado no apartado anterior. Este simulador é implementado como un "driver" que mantén unha conexión cun servidor TCP remoto e mediante o que poden simularse calquera número de entradas e saídas. A Figura 7.23 mostra o diagrama das clases utilizadas.

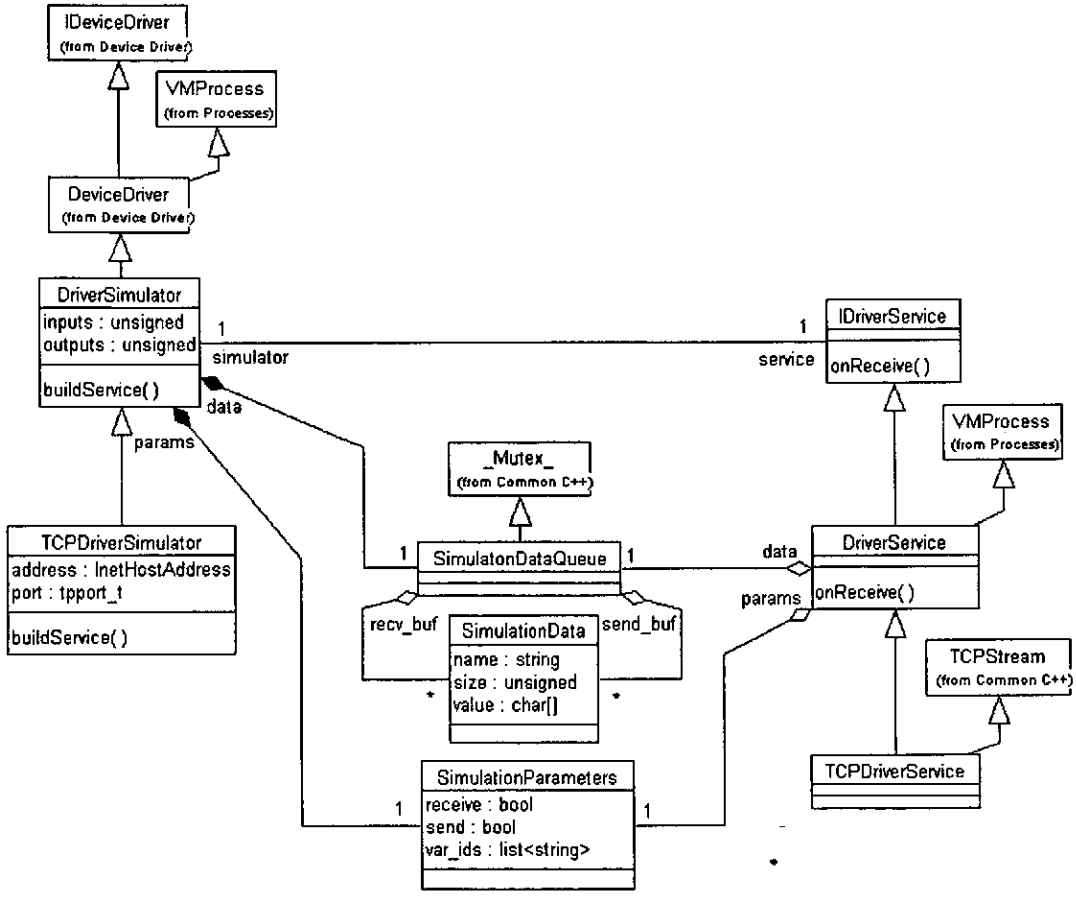


Figura 7.23. Diagrama das clases utilizadas para a implementación dun "driver" TCP/IP de simulación.

A clase *DriverSimulator*, derivada de *DeviceDriver* (§7.2.1), é a que implementa o simulador dos dispositivos de E/S. O constructor desta clase recibe como parámetro o número de entradas e saídas a simular, polo que as súas instancias permiten simular dispositivos de diferentes tamaños no que a entradas e saídas se refire. A conexión do "driver" co mediador remoto é representada mediante a clase abstracta *IDriverService*. Para manter a implementación do "driver" independente do sistema de comunicación utilizado, na clase *DriverSimulator* declárase un método constructor virtual ("factory method" [67]), que será redefinido nas clases derivadas e que devolverá unha instancia dunha clase derivada da interface *IDriverService*. Por exemplo, a clase *TCPDriverSimulator* devolve unha instancia da clase *TCPDriverService* que é un proceso da máquina virtual que xestiona unha conexión cun servidor TCP remoto.

A clase *DriverService* proporciona a implementación por defecto dun proceso da máquina virtual que xestione unha conexión cun mediador remoto e a comunicación co simulador do dispositivo de E/S. Esta comunicación realízase mediante o intercambio de valores a través de

colas e a notificación asíncrona (§7.1.2.4.2) da recepción de valores na conexión —método *onReceive*—. As colas de valores recibidos e os pendentes de enviar son almacenados polas instancias da clase *DriverSimulator* —agregación *data*—, xunto cos parámetros de configuración a enviar ao mediador no establecemento da conexión —agregación *params*—. As instancias da clase *TCPDriverSimulator* reciben apuntadores a esta información como argumentos do método *buildService* (Figura 7.23) cando é establecida a conexión co mediador remoto.

A Figura 7.24 mostra a secuencia de mensaxes intercambiadas entre o simulador de dispositivo, o proceso que xestiona a conexión co mediador e o propio mediador para o establecemento da conexión e o intercambio de valores de simulación.

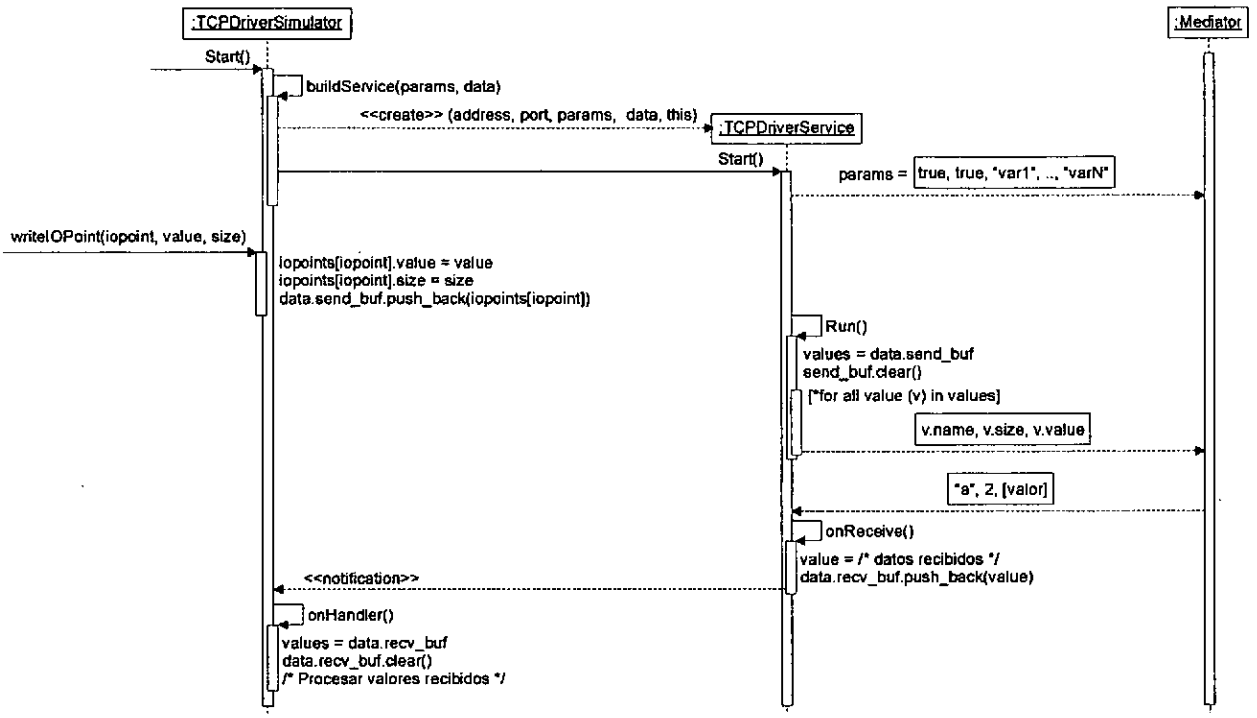


Figura 7.24. Secuencia de mensaxes do intercambio de información de simulación utilizando un "driver" TCP/IP.

7.2.3.4.1. Implementación da interface IDeviceDriver

Dos métodos da interface *IDeviceDriver*, a clase *DriverSimulator* implementa os utilizados para crear a información de configuración do dispositivo (§7.2.1.2) e ler e escribir valores (§7.2.1.3); e redefine os implementados pola clase *DeviceDriver* para xestionar a asignación de variábeis a puntos de E/S (§7.2.1.4). A continuación danse algúns detalles sobre a implementación destes métodos.

Creación da información de configuración do dispositivo

A implementación do método *buildDeviceInfo* crea unha información de configuración que define un dispositivo formado por dous subsistemas, denominados *input_channels* e *output_channels* respectivamente. O primeiro conterá as definicións dos puntos de entrada e o segundo as de saída. En cada subsistema engadirase un número de puntos igual ao valor dos atributos *inputs* e *outputs* iniciados no constructor do simulador do dispositivo. Os identificadores dos puntos serán *input_channel_N* para as entradas e *output_channel_N* para as saídas. Polo tanto o identificador completo dun punto de entrada seria

'*\\ddsim\iodevice\input_channels\input_channel_n*'. Na versión actual do simulador non se define ningunha característica asociada aos subsistemas ou aos puntos de E/S. En futuras versións poderían utilizarse, por exemplo, para restrinxir o tipo de datos das variábeis asignadas aos puntos de E/S.

Asignación de variábeis a puntos de E/S

Como se mostra na Figura 7.25, a clase *DriverSimulator* mantén internamente dous mapas, un con información sobre as variábeis asignadas a puntos de E/S —agregación *vars*—, e outro cos últimos valores recibidos ou enviados en cada punto de E/S que teña unha variábel asignada —agregación *iopoints*—. Os métodos que modifican a asignación de variábeis a puntos de E/S (*scheduleInput*, *scheduleOutput*, *unscheduleInput*, *unscheduleOutput*, e *unscheduleAll*) son redefinidos de xeito que se manteñan correctamente actualizados estes mapas cada vez que se realice ou elimine unha asignación.

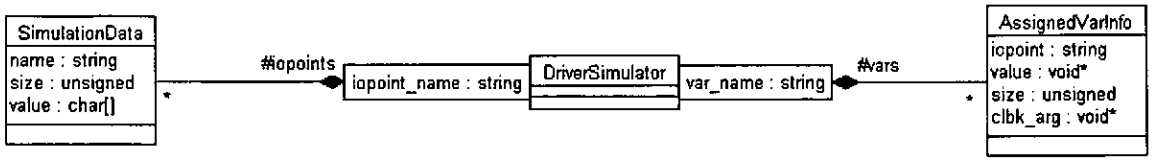


Figura 7.25. Diagrama de clases para a asignación de variábeis no "driver" de simulación.

Lectura e escritura de valores

Os métodos de lectura e escritura de valores non acceden directamente á conexión co mediador remoto, senón que se utilizan os datos almacenados nos mapas do simulador (Figura 7.25) para implementar a lectura síncrona e asíncrona e as colas de comunicación coa conexión (Figura 7.23) para a escritura de valores. O pseudocódigo seguinte mostra a implementación do método *listenIOPoint*, que almacena no mapa *vars* a información precisa para realizar unha notificación asíncrona cando se reciba un valor no punto de E/S indicado:

```
2504. DriverSimulator::listenIOPoint(const string& iopoint_name,
2505.                                 void* value,
2506.                                 unsigned size,
2507.                                 pfn callback,
2508.                                 IAsyncClbkArg* clbk_arg)
2509. {
2510.   // obter variábel asignada ao punto de E/S
2511.   iopoint_info = iopoints[iopoint_name]
2512.   var_info = vars[iopoint_info->name]
2513.   // almacenar información para a notificación asíncrona
2514.   var_info.clbk_arg = clbk_arg
2515.   var_info.value = value
2516.   var_info.size = size
2517. }
```

Os métodos *stopListeningIOPoint* e *receivedIOPoint* tamén acceden á información da variábel asignada ao punto de E/S indicado, que é almacenada no mapa *vars*, para iniciala e para consultar se o valor do atributo *value* foi modificado, respectivamente. O procesamento dunha lectura asíncrona complétase cando se recibe un valor para a variábel asignada na porta de E/S. Cando isto acontece, a notificación asíncrona recibida dende o proceso que xestiona a conexión co mediador (Figura 7.24) é manexada no método *onHandler* (liña 2033) do simulador, que ten o pseudocódigo seguinte:

```
2518. DriverSimulator::OnHandler(void* arg)
2519. {
2520.   // copiar valores recibidos da cola de entrada a un "buffer" local
2521.   data.EnterMutex()
2522.   sim_values = data.recv_buf
2523.   data.recv_buf.clear()
2524.   data.LeaveMutex()
2525.   // almacenar valores recibidos no mapa iopoints
2526.   for each sim_value in sim_values
2527.     var_info = vars[sim_value.name]
2528.     iopoints[var_info.iopoint] = sim_value
2529.     // realizar notificación asíncrona se estivera activada
2530.     if (var_info.clbk_arg)
2531.       var_info.value = sim_value.value
2532.       DeviceDriver::OnHandler(var_info.clbk_arg)
2533.     end if
2534.   end for
2535. }
```

A implementación deste método comeza copiando os valores recibidos dende o mediador (que estarán almacenados na cola *recv_buf*) a un "buffer" local. Despois, cada valor recibido é almacenado no mapa *iopoints*, no elemento correspondente ao punto de entrada no que a variábel estea asignada. Por último, se estivera activada a lectura asíncrona da variábel, copiase o valor recibido ao "buffer" pasado como argumento na chamada ao método *listenIOPoint*. A notificación é procesada no método *onHandler* da clase *DeviceDriver*, que envía unha mensaxe co valor recibido á porta de saída do simulador de dispositivos.

No caso de non estar activada a lectura asíncrona, o valor recibido será recuperado cando se invoque o método *readIOPoint*. O pseudocódigo deste método é o seguinte:

```
2536. DriverSimulator::readIOPoint(const string& iopoint_name,
2537.                              void* value,
2538.                              unsigned size)
2539. {
2540.   // recuperar valor almacenado no mapa iopoints
2541.   iopoint_info = iopoints[iopoint_name]
2542.   value = iopoint_info.value
2543. }
```

No referente á escritura de valores, a implementación do método *writeIOPoint* almacena o valor a enviar no mapa *iopoints* e na cola *send_buf*, da que serán lidos polo proceso que xestiona a conexión co mediador. O pseudocódigo deste método é o seguinte:

```
2544. DriverSimulator::writeIOPoint(const string& iopoint_name,
2545.                               void* value,
2546.                               unsigned size)
2547. {
2548.   // almacenar valor no mapa iopoints
2549.   iopoint_info = iopoints[iopoint_name]
2550.   iopoint_info.value = value
2551.   iopoint_info.size = size
2552.   // copiar valor na cola de saída
2553.   data.EnterMutex()
2554.   data.send_buf.push_back(iopoints[iopoint_name])
2555.   data.LeaveMutex()
2556. }
```

7.3. O núcleo da máquina virtual

O núcleo da máquina virtual (Figura 7.8) está formado pola base de datos de E/S, os procesos que xestionan os temporizadores e o depurador. Nesta sección se explican os detalles do deseño e implementación da base de datos de E/S (§7.3.1), os temporizadores (§7.3.2) e os servicios que o núcleo proporciona ás aplicacións executadas no subsistema de aplicación (§7.3.3). Os detalles do depurador non se incluíron nesta documentación —na versión actual a súa única función é o envío, a través dun enlace de comunicación, da situación do modelo Grafcet en execución utilizando o formato explicado en (§F.4.5)—.

7.3.1. A base de datos de E/S

A base de datos é o proceso da máquina virtual que almacena os valores das variábeis do proceso, xa sexan estes simulados ou obtidos mediante "drivers" de E/S. Como pode verse na Figura 7.26, a base de datos dispón de catro portas de enlace a través das que recibe e envía mensaxes aos xestores de E/S (§7.2.2) e ao subsistema de aplicación (§7.5).

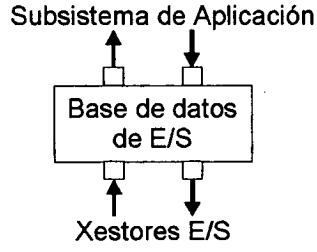


Figura 7.26. Estructura da base de datos de E/S.

7.3.1.1. Funcionalidades da base de datos de E/S

As funcionalidades que a base de datos de E/S proporciona son as seguintes:

1. Mantén un rexistro histórico dos valores das variábeis. Cada valor é almacenado xunto coa hora e data na que se obtén e pode configurarse o número de valores históricos a almacenar para cada variábel.
2. Contén a imaxe actual do proceso, consistente en dúas táboas cos valores das entradas e saídas que unicamente se actualizan cando se recibe unha petición de sincronización dende o subsistema de aplicación. As mensaxes recibidas pola base de datos contendo novos valores para as variábeis do proceso unicamente afectan á imaxe do proceso cando se realice esa sincronización.
3. Mantén un "buffer" de entrada e outro de saída, nos que as mensaxes son almacenadas ata que se reciba unha petición de sincronización co proceso.
4. Encárgase de xerar os eventos que indican os cambios de valor nas entradas booleanas do proceso.

7.3.1.2. Implementación da base de datos de E/S

O diagrama de clases da Figura 7.27 mostra as clases e as relacións utilizadas na implementación da base de datos. A clase principal é a denominada *IORTDB*, derivada de *VMProcess*, que contén dúas colas de mensaxes —agregacións *input_messages* e *output_messages*—, os valores das variábeis do proceso —agregacións *inputs* e *outputs*— e as imaxes do proceso utilizadas durante a evolución dos modelos executados no subsistema de aplicación —agregacións *input_image* e *output_image*—. As imaxes do proceso son

representadas mediante instancias da clase *IORTDBSnapshot*, que é unha colección de variábeis obtida por especialización da clase parametrizada *pdcid2ptrSeq* (§B.1). Do mesmo xeito as variábeis de proceso son almacenadas nunha instancia da clase *IORTDBEntrySeq*, que é unha colección de instancias derivadas da interface *IIORTDBEntry*, obtida igualmente por especialización da clase parametrizada *pdcid2ptrSeq*.

A clase *IORTDBEntry*, derivada da interface *IIORTDBEntry*, é unha clase parametrizada que representa as entradas da base de datos. As instancias desta clase son parametrizadas co tipo de datos da variábel que almacenan. Haberá unha entrada por cada variábel contida na base de datos. Para cada variábel almacenase o seu valor actual —agregación *process_var*—, o rexistro de valores históricos —agregación *history_values*—, os identificadores alfanuméricos do "driver" e do punto de E/S no que a variábel estea asignada —atributos *ddriver_id* e *iopoint_id*, respectivamente—, e o numero de valores históricos a almacenar —atributo *num_values*—. O valor actual e os valores históricos son representados mediante instancias obtidas por especialización dos "templates" *GescaSystemVar* e *TimedValue* (§B.2), utilizando como argumento o tipo de datos da variábel almacenada.

O código seguinte mostra unha versión simplificada da declaración da interface pública da clase *IORTDB*:

```
2557. class IORTDB : public VMProcess
2558. {
2559. public:
2560.   // engadir e eliminar entradas da BD
2561.   bool isEmpty() const;
2562.   bool insert(ISystemDataDeclaration* var_decl);
2563.   bool remove(const string& var);
2564.   bool removeAll();
2565.   // sincronización co proceso
2566.   bool getInputEvents(MsgBuffer& events);
2567.   bool synchronizeInputs();
2568.   bool synchronizeOutputs();
2569.   // consulta e modificación de valores
2570.   template <class T>
2571.    bool getInputSnapshot(const string& var, T& val);
2572.   template <class T>
2573.    bool setOutputSnapshot(const string& var, const T& val);
2574.   template <class T>
2575.    bool getIORTDBValue(const string& var, T& val);
2576.   template <class T>
2577.    bool setIORTDBValue(const string& var, const T& val);
2578.   template <class T>
2579.    bool getIORTDBTimedValue(const string& var,
2580.                             unsigned span, // valor histórico, 0 = actual
2581.                             TimedValue<T>& tval);
2582.   template <class T>
2583.    bool setIORTDBTimedValue(const string& var, const TimedValue<T>& tval);
2584. };
```

Esta interface está composta polos métodos que permiten engadir e eliminar entradas na base de datos (liñas 2562-2564), sincronizar a imaxe do proceso cos valores actuais das variábeis (liñas 2566-2568), e acceder e modificar tanto os valores da imaxe do proceso (liñas 2570-2573) coma os valores actuais (liñas 2574-2577) e históricos (liñas 2578-2583) das variábeis.

A Figura 7.28 mostra a secuencia de mensaxes intercambiadas para a creación dunha nova entrada na base de datos (implementación do método *insert*). A nova entrada é creada utilizando o método *CreateIORTDBEntry*, un método constructor ("factory method" [67]) ao

que se accede a través da instancia da clase parametrizada *GescaSystemVarDecl* (§B.2) pasada como parámetro da chamada ao método *insert*. Esta instancia contén a información da variábel de proceso a almacenar na base de datos. Unha vez creada a nova entrada esta é almacenada na base de datos como parte das entradas ou das saídas dependendo da declaración da variábel (son consideradas como entradas as variábeis que devolven un valor *true* na chamada ao método *canRead*).

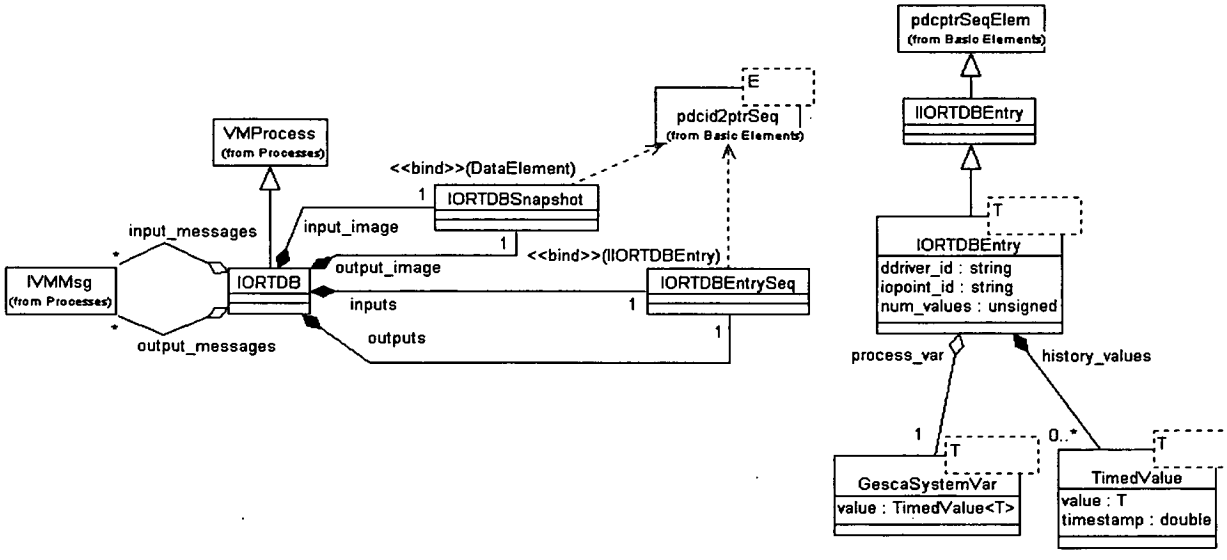


Figura 7.27. Diagrama das clases que modelan a base de datos de E/S.

A declaración simplificada da interface pública da clase *IIORTDBEntry* é a seguinte:

```
2585. class IIORTDBEntry : public pdcptrSeqElem
2586. {
2587. public:
2588.   // consulta da información da entrada
2589.   virtual string getDeviceID(void) const = 0;
2590.   virtual string getIOPointID(void) const = 0;
2591.   virtual unsigned getNumValues(void) const = 0;
2592.   virtual type_info typeOf() const = 0;
2593.   virtual size_t sizeOf() const = 0;
2594.   // consulta e modificación de valores
2595.   template <class T>
2596.     void getValue(T& val);
2597.   template <class T>
2598.     void setValue(const T& val);
2599.   template <class T>
2600.     void getTimedValue(unsigned span, // valor histórico, 0 = actual
2601.                        TimedValue<T>& tval);
2602.   template <class T>
2603.     void setTimedValue(const TimedValue<T>& tval);
2604. };
```

A interface está formada polos métodos que permiten acceder á información sobre a variábel almacenada na entrada da base de datos (liñas 2589-2593): tipo, tamaño, número de valores históricos, e "driver" e punto de E/S no que está asignada; e acceder e modificar tanto o valor actual (liñas 2595-2598) coma os históricos (liñas 2599-2603) da variábel.

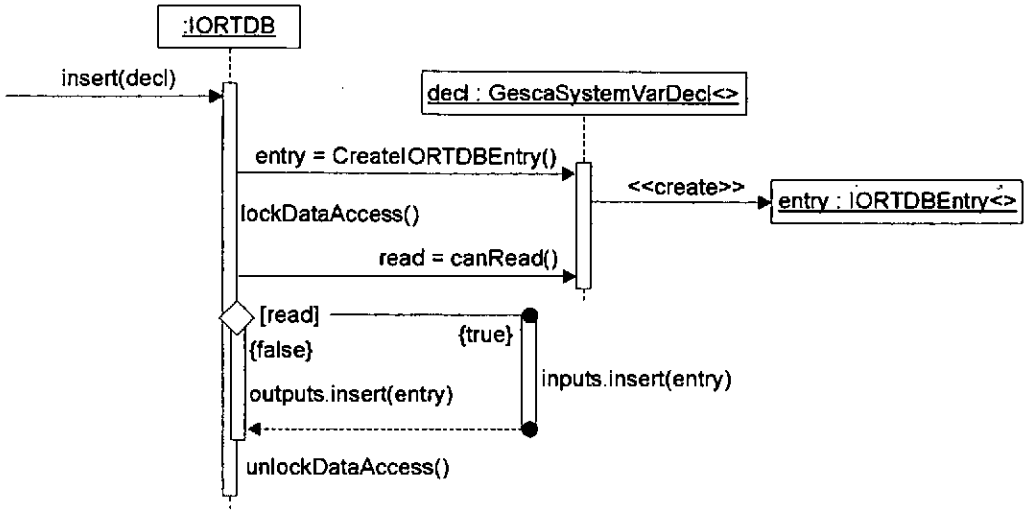


Figura 7.28. Secuencia das mensaxes intercambiadas na creación dunha nova entrada na BD.

7.3.1.3. Actualización de valores e sincronización da imaxe do proceso

O valor dunha variábel almacenada na base de datos é actualizado cando se recibe unha mensaxe contendo un novo valor. O diagrama da Figura 7.29 mostra a secuencia de mensaxes intercambiadas para a xestión dos valores recibidos dende o xestor de entradas (§7.2.2). A secuencia comeza coa solicitude dunha lectura sen bloqueo na porta de enlace pola que se reciben as mensaxes do xestor de entradas. Cando recibe unha mensaxe, a porta de enlace envía unha notificación asíncrona á base de datos que é manexada no seu método *OnHandler* (liña 2033). A mensaxe recibida conterá o identificador da variábel, o seu novo valor e a data e hora na que foi adquirido polo "driver" de entrada no que estivera asignada. O procesamento da mensaxe consiste en localizar a entrada que se corresponde coa variábel modificada e actualizar o seu valor. Isto realízase na implementación do método *setTimedValue* (liña 2603), que se encarga tanto da modificación do valor actual da variábel como da correcta xestión dos seus valores históricos. Ademais, se a variábel modificada é booleana e o valor actual é diferente do valor que tiña anteriormente, a base de datos crea e almacena o evento que indica este cambio. Estes eventos serán enviados ao subsistema de aplicación cada vez que este solicite unha sincronización co proceso. O procesamento remata coa destrucción da mensaxe recibida e o inicio dunha nova operación de lectura sen bloqueo na porta de enlace que conecta a base de datos co xestor de entradas.

A sincronización dos valores de entrada almacenados na imaxe do proceso (Figura 8.6) e os da base de datos realízase implicitamente cando se invoca o método *getInputEvents* (liña 2566) ou explicitamente co método *synchronizeInputs* (liña 2567). O algoritmo utilizado para realizar a sincronización actualiza unicamente os valores da imaxe que foron modificados na base de datos dende a última sincronización (o indicador booleano *modified* é utilizado para indicar esta circunstancia). O seu pseudocódigo é o seguinte:

```
2605. for each var V in input_image
2606.    entry = inputs[V]
2607.    if inputs[V].modified
2608.       input_image[V] = inputs[V][0] // almacenar o valor actual na imaxe
2609.       inputs[V].modified = false
2610.    end if
2611. end for
```

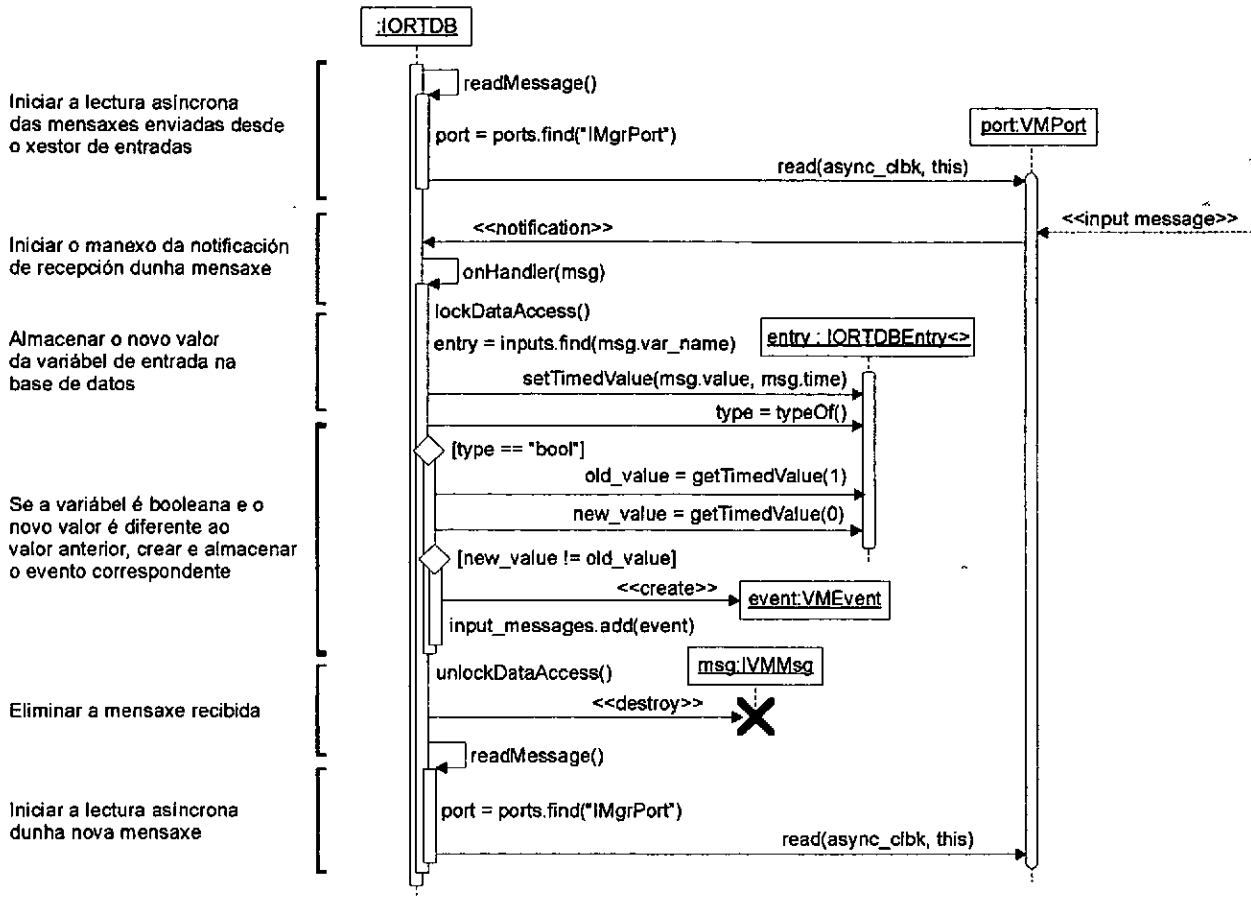


Figura 7.29. Secuencia das mensaxes intercambiadas no procesamento das mensaxes recibidas dende o subsistema de E/S.

O proceso utilizado para manexar as mensaxes de saída enviadas dende o subsistema de aplicación é semellante ao explicado para as entradas. A principal diferencia consiste en que para cada mensaxe recibida actualízase o valor da saída na imaxe do proceso. A sincronización da imaxe das saídas co proceso (Figura 8.9), que consiste na actualización dos valores das saídas almacenados na base de datos e o envío das mensaxes ao xestor de saídas cos valores modificados, realízase explicitamente cando se invoca o método *synchronizeOutputs* (liña 2568).

7.3.2. Temporizadores

A implementación da máquina virtual inclúe dous tipos diferentes de temporizadores que son utilizados para as funcións de control do tempo que os procesos da súa arquitectura requiran, así como para as temporizacións especificadas nos modelos executados no subsistema de aplicación (§7.5). Todos os temporizadores da máquina virtual son implementados como procesos que proporcionan servicios de temporización aos que se lles asigna a prioridade máis alta. Debe terse en conta que a resolución e precisión dos temporizadores dependen do soporte proporcionado pola combinación "hardware/software" na que se implemente a máquina virtual.

7.3.2.1. Temporizador simple

O primeiro tipo de temporizador definido proporciona un servicio básico de temporización en dúas versións:

1. *Con bloqueo*, o proceso que solicita o servicio é bloqueado durante o tempo indicado.
2. *Con notificación de finalización*, o proceso que solicita o servicio recibe unha notificación asíncrona (§7.1.2.4.2) cando transcorre o período de tempo indicado. Ademais, nesta versión, pode solicitarse a repetición da notificación cunha periodicidade determinada.

Os temporizadores deste tipo implementan a interface *ITimerServer*:

```
2612. struct ITimerServer
2613. {
2614.  // control do estado do temporizador
2615.  virtual void Start() = 0;
2616.  virtual void Suspend() = 0;
2617.  virtual void Resume() = 0;
2618.  virtual void Finish() = 0;
2619.  virtual bool isSuspended() = 0;
2620.  virtual bool isRunning() = 0;
2621.  // servicios de temporización
2622.  virtual bool set(long time,
2623.                   long period = 0,
2624.                   pfn clbk = NULL,
2625.                   IAsyncClbkArg* clbkarg = NULL) = 0;
2626.  virtual bool wait(long time) = 0;
2627.  virtual void stop(void) = 0;
2628. };
```

A interface declara métodos comúns aos da interface *IVMProcess* (§7.1.2.1) para o control e consulta do estado do temporizador. O método *wait* (liña 2626) proporciona o servicio de temporización con bloqueo e o método *set* (liña 2622) o de temporización con notificación. Ambos métodos reciben como argumento a cantidade de tempo a considerar. O método *set* recibe ademais os argumentos relacionados coa notificación asíncrona, entre eles un valor opcional utilizado para as notificacións periódicas. A interface complétase co método *stop* (liña 2627) que detén o temporizador. A Figura 7.30 mostra a secuencia de mensaxes das diferentes opcións de temporización comentadas.

7.3.2.2. Temporizador múltiple

O tipo de temporizador explicado anteriormente non pode atender máis dunha solicitude simultaneamente. Debido a que os temporizadores son recursos limitados nun sistema e, en ocasións, é preciso ter activas un grande número de temporizacións simultaneamente, definiuse un segundo tipo de temporizador que permite a planificación con notificación de múltiples temporizacións. A cada unha se lle asigna un identificador numérico que é incluído na notificación e que pode ser utilizado para deter individualmente cada temporización. En contrapartida este temporizador introduce un certo "overhead" debido á xestión das temporizacións que pode non ser aceptábel en sistemas que requiran unha precisión moi alta. Estes temporizadores implementan a interface *IMultitimerServer*:

```
2629. struct IMultitimerServer
2630. {
2631.  // control do estado do temporizador
2632.  virtual void Start() = 0;
2633.  virtual void Suspend() = 0;
2634.  virtual void Resume() = 0;
2635.  virtual void Finish() = 0;
2636.  virtual bool isSuspended() = 0;
2637.  virtual bool isRunning() = 0;
```

```
2638.  // servicios de temporización
2639.  virtual bool doSchedules(void) = 0;
2640.  virtual bool doUnschedules(void) = 0;
2641.  virtual unsigned schedule(long time,
2642.                            pfn clbk,
2643.                            IAsyncClbkArg* clbkarg = NULL,
2644.                            bool deferred = false) = 0;
2645.  virtual bool unschedule(unsigned id, bool deferred = false) = 0;
2646.  virtual void stop(void) = 0;
2647.};
```

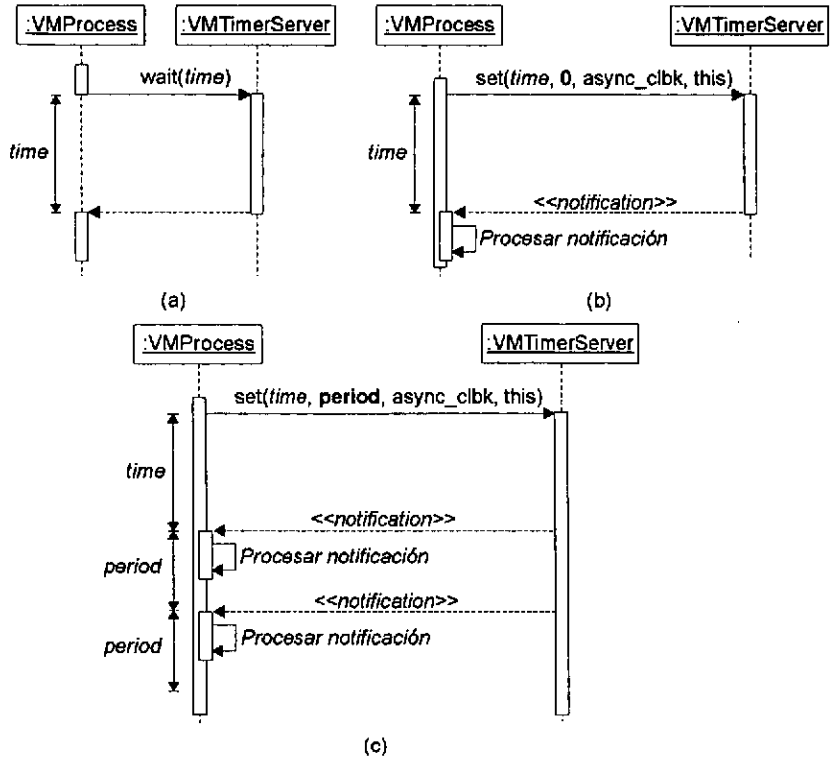


Figura 7.30. Servicios proporcionados polo temporizador básico: a) temporización con bloqueo; b) temporizador con notificación asíncrona; e c) temporizador con notificación periódica.

Do mesmo xeito que na interface *ITimerServer*, decláranse métodos comúns aos da interface *IVMProcess* para o control e consulta do estado do temporizador. A planificación dun novo temporizador faise co método *schedule* (liña 2641), que recibe como argumento a cantidade de tempo a considerar, os argumentos utilizados para a notificación asíncrona e un valor booleano que indica se a planificación debe terse en conta inmediatamente ou permanecer en suspenso ata que sexa invocado o método *doSchedules* (liña 2639). O método *schedule* devolve o identificador numérico asignado ao temporizador. O método *unschedule* (liña 2645) permite deter a planificación dun temporizador concreto indicado mediante o seu identificador. Do mesmo xeito que acontecía coa planificación, este método recibe como argumento un valor booleano que indica se a parada do temporizador debe facerse inmediatamente ou manterse en suspenso ata que sexa invocado o método *doUnschedules* (liña 2640). A interface complétase co método *stop* (liña 2646) que detén todos os temporizadores planificados.

A Figura 7.31 mostra un exemplo da secuencia de mensaxes intercambiadas entre dous procesos da máquina virtual e un temporizador que implemente a interface *IMultitimerServer*. Un dos procesos inicia tres temporizadores que inicialmente estarán suspendidos ata que se invoque o método *scheduleAll*. O outro proceso inicia dous temporizadores, un deles suspendido e o outro non —*timer_4*—. Unha vez transcorrido o tempo indicado —*time_4*— o

proceso que iniciou o temporizador recibe e procesa a notificación correspondente. Os demais temporizadores son activados na chamada a *scheduleAll*. Unha vez transcorrido o tempo indicado para cada un deles o proceso que os iniciou recibe e procesa a notificación correspondente. Unha excepción é o temporizador *timer_1* que é desactivado polo método *unschedule* antes de que transcorra o tempo *time_1*. Nótese que tamén se fai unha chamada a este método para o temporizador *timer_3*, mais neste caso é unha desactivación que queda suspendida ata a seguinte chamada ao método *unscheduleAll*. Como isto non acontece antes de que transcorra o tempo *time_3* o temporizador non é desactivado. Sen embargo o temporizador *timer_6*, que está en estado suspendido cando se chama a *unscheduleAll*, si é desactivado e nunca chega a iniciarse.

7.3.3. Acceso aos servicios do núcleo

O acceso aos servicios proporcionados polo núcleo da máquina virtual realízase mediante unha clase que implemente a interface *IVMServices* que sirve como punto de acceso común ao núcleo (patrón "Facade" [67]). A declaración da interface *IVMServices* é a seguinte:

```
2648.class IVMServices
2649.{
2650.public:
2651.  // temporizadores
2652.  virtual bool doSchedules(void) = 0;
2653.  virtual bool doUnschedules(void) = 0;
2654.  virtual unsigned long schedule(long time,
2655.                                 pfn callback,
2656.                                 IAsyncClbkArg* clbk_arg = NULL,
2657.                                 bool deferred = true) = 0;
2658.  virtual bool unschedule(unsigned long _id, bool deferred = true) = 0;
2659.  // eventos/entradas/saídas
2660.  virtual bool getInputEvents(deque<VMEvent*>& events) = 0;
2661.  virtual bool setOutputValues() = 0;
2662.  template <class T>
2663.    bool getSystemData(const string& id, T& value);
2664.  template <class T>
2665.    bool setSystemData(const string& id, const T& value);
2666.  // xestión de erros
2667.  virtual void onError(unsigned code, IModule* source) = 0;
2668.};
```

Como pode verse, os servicios do núcleo aos que pode accederse utilizando esta interface son os seguintes:

1. A xestión de temporizacións.
2. O acceso aos eventos almacenados no canal de entrada do subsistema de aplicación e aos valores da base de datos de E/S, así como a actualización nesta dos valores de saída modificados no subsistema de aplicación.
3. A sinalización de erros detectados durante a execución.

7.4. O subsistema de xestión da configuración

A máquina virtual proporciona un conxunto de servicios para a xestión da súa configuración e o control do seu funcionamento que poden ser accedidos remotamente mediante o intercambio de mensaxes simples. O acceso a estes servicios foi deseñado para ser independente do medio de transmisión utilizado mediante a definición de interfaces que abstraen as características específicas de cada sistema e que haberá que implementar en cada caso concreto. A máquina virtual inclúe unha implementación por defecto para redes de

comunicación que utilicen o protocolo TCP/IP. En futuras versións poderían proporcionarse implementacións que utilicen outros medios como liñas serie de transmisión de datos, protocolos distribuídos de intercambio de datos entre procesos, etc. Unha limitación da versión actual da máquina virtual é que non acepta máis dunha conexión remota simultánea. No resto deste apartado descríbese o formato e o protocolo de intercambio de mensaxes, os servicios proporcionados e a implementación do acceso remoto en redes TCP/IP.

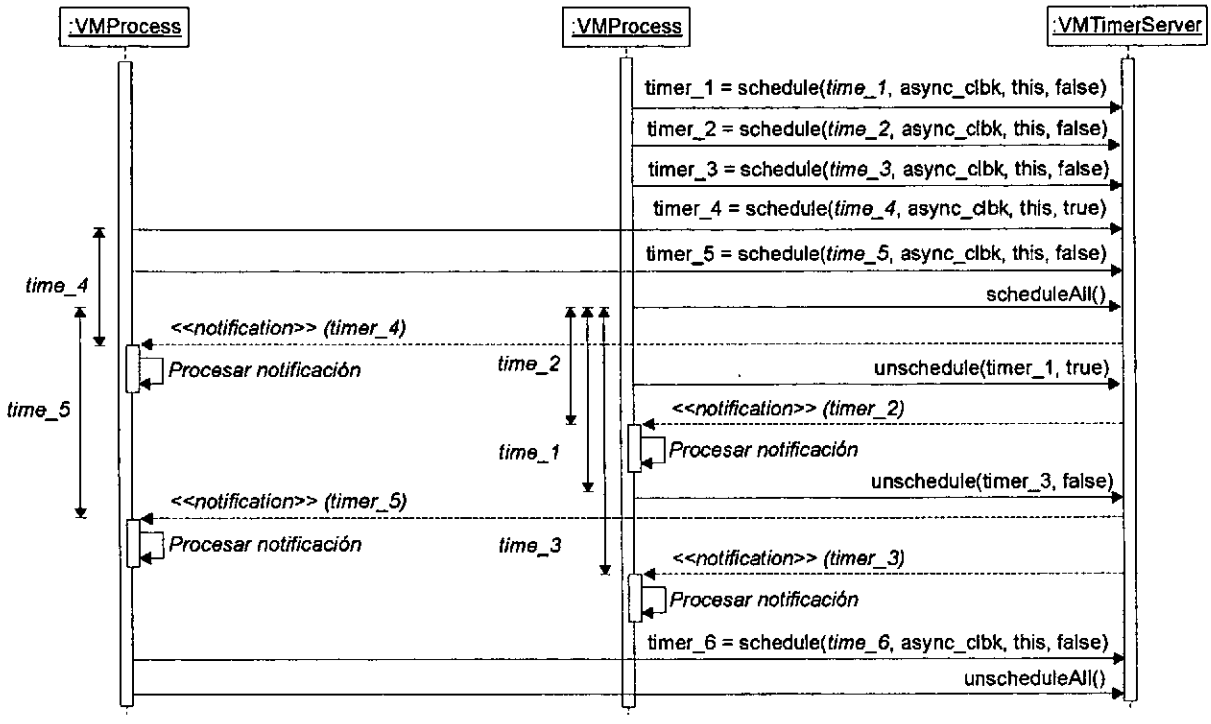


Figura 7.31. Secuencia de mensaxes intercambiadas na planificación de temporizadores.

7.4.1. Formato das mensaxes

A Figura 7.32 mostra o formato das mensaxes utilizadas no acceso remoto aos servicios da máquina virtual. Cada mensaxe está formada por unha cabeceira e un conxunto opcional de parámetros de tamaño variábel. A cabeceira contén tres campos:

1. O código que identifica o tipo de mensaxe (4 bytes).
2. Un código auxiliar (4 bytes) utilizado para indicar condicións excepcionais, como por exemplo un código de erro.
3. O número de parámetros opcionais (4 bytes).

Pola súa banda cada parámetro opcional está formado por dous campos:

1. O tamaño en bytes do parámetro (4 bytes).
2. O valor do parámetro.

Tendo en conta os valores dos campos da cabeceira dunha mensaxe, estas poden clasificarse en tres tipos diferentes (Figura 7.33):

1. *As mensaxes de control*, utilizadas para indicar a finalización correcta dun servicio (mensaxe OK!) ou a detección dunha condición de erro (mensaxe FAIL).

2. *As mensaxes de resposta*, utilizadas pola máquina virtual para enviar datos. O valor do código destas mensaxes é OK, e o seu código auxiliar é maior ou igual a un.
3. *As ordes*, mensaxes utilizadas polo cliente que accede remotamente á máquina virtual para solicitar un servicio. As ordes poden ser simples, formadas por unha soa mensaxe, ou múltiples, formados por varias mensaxes. Neste caso utilízase a mensaxe que delimita o final dunha orde múltiple (mensaxe CMD_END) para indicar cal é a última mensaxe da orde.

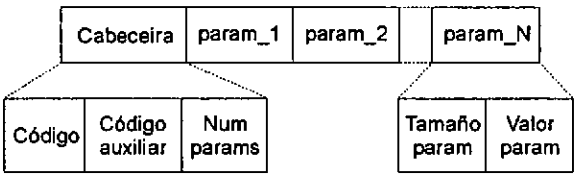


Figura 7.32. Formato das mensaxes utilizadas no acceso remoto aos servicios da máquina virtual.

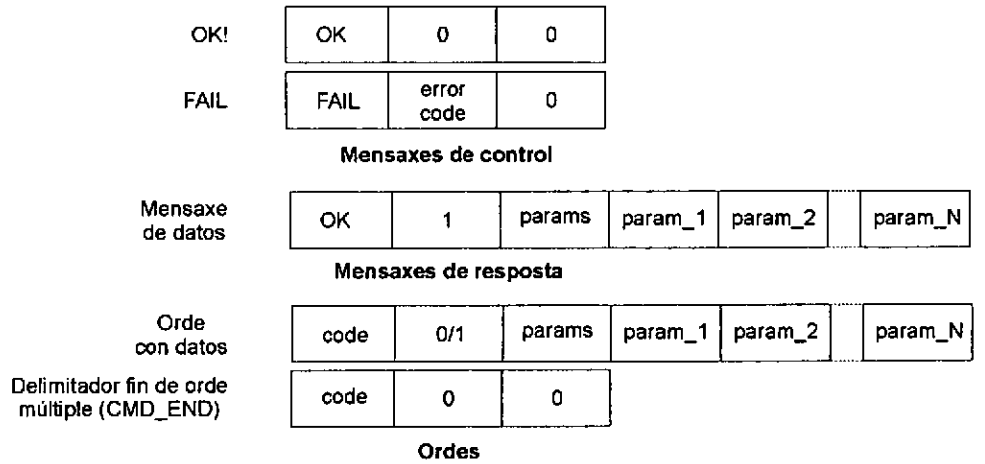


Figura 7.33. Tipos de mensaxes utilizadas no acceso remoto aos servicios da máquina virtual.

7.4.2. Protocolo de intercambio de mensaxes

Coas mensaxes descritas no apartado anterior son varios os patróns de comunicación definidos para dar soporte ao intercambio de información que se produce no acceso aos servicios remotos ofrecidos pola máquina virtual. A definición destes patróns faise partindo da suposición de que o medio de comunicación utilizado é fiábel (non perde mensaxes) e mantén a orde de envío das mensaxes. Tendo en conta estas suposicións, poden producirse as seguintes interaccións entre un cliente e a máquina virtual:

1. Petición, cunha orde simple, dun servicio que non devolva resposta. Se o servicio pode realizarse a máquina virtual devolverá unha mensaxe OK! (Figura 7.34.a), en caso contrario devolverá unha mensaxe FAIL (Figura 7.34.b) cun código que indique o motivo. Nótese que a orde pode conter parámetros ou non e que o valor do seu código auxiliar é cero.
2. Petición, cunha orde simple, dun servicio que devolva resposta (Figura 7.34.c). Neste caso a máquina virtual envía múltiples mensaxes resposta contendo os datos resultado do servicio. Nótese que o valor do código auxiliar nestas mensaxes é un, e que o número de parámetros e maior ou igual a un. A resposta da máquina virtual remata cunha mensaxe OK!. En caso de producirse un erro, a máquina virtual enviaría a mensaxe FAIL e a operación quedaría abortada.

3. Petición, cunha orde múltiple, dun servicio que non devolva resposta (Figura 7.34.d). O cliente envía múltiples ordes co mesmo código. Cada unha destas ordes poderá conter parámetros ou non, mais o valor do seu código auxiliar será maior ou igual a un. O final da orde múltiple se indica co envío dunha mensaxe CMD_END. A máquina virtual devolverá OK! ou FAIL despois de cada mensaxe recibida, dependendo do resultado da comunicación e da execución do servicio.
4. Petición, cunha orde múltiple, dun servicio que devolva resposta. Este caso é unha combinación dos dous anteriores.

Nótese que a máquina virtual unicamente envía mensaxes con códigos OK ou FAIL e que o cliente sempre agarda a recibir unha destas mensaxes antes de enviar a súa seguinte mensaxe.

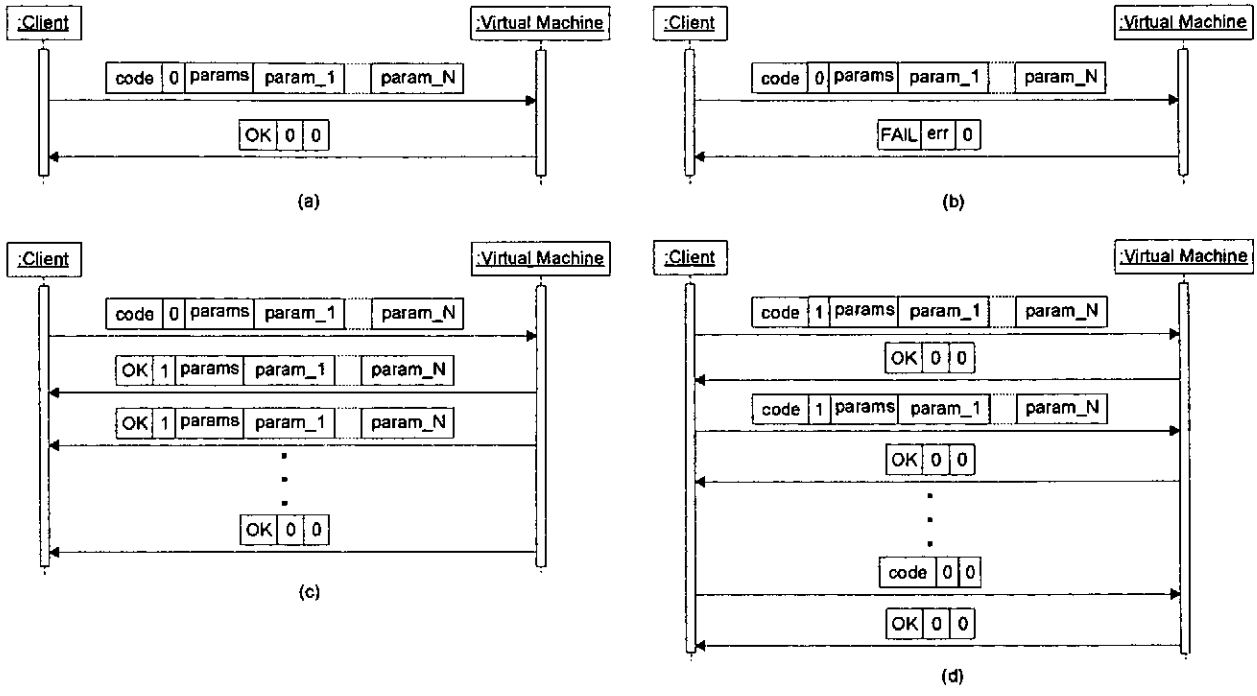


Figura 7.34. Protocolo de intercambio de mensaxes no acceso remoto aos servicios da máquina virtual: a) orde simple sen resposta (resultado correcto); b) orde simple sen resposta (resultado erróneo); c) orde simple con resposta; e d) orde múltiple sen resposta (resultado correcto).

7.4.3. Servicios remotos da máquina virtual

Os servicios proporcionados pola máquina virtual a través do mecanismo de acceso remoto poden clasificarse nas categorías seguintes:

1. *Carga e descarga de DLLs*, nesta categoría agrúpanse os servicios que permiten:
   a. Iniciar remotamente a carga en memoria dos módulos contidos nunha DLL, utilizando o mecanismo explicado en (§7.1.1.5).
   b. Descargar unha DLL e os seus módulos da memoria da máquina virtual.
   c. Descargar todas as DLLs e módulos da memoria da máquina virtual.
2. *Carga e descarga de táboas de E/S* (táboas que conteñen as declaracións das variábeis do proceso e a información da súa asignación aos "drivers" de E/S), nesta categoría agrúpanse os servicios que permiten:

a. Iniciar remotamente a carga en memoria dunha táboa de E/S. Estas táboas conteñen a información sobre as variábeis de proceso utilizadas. Para cada variábel indícase o seu nome, tipo e tamaño; o número de valores históricos a almacenar na base de datos (§7.3.1); e a información utilizada para asignar a variábel a un dispositivo de E/S (§7.2.1.4.1): identificador do "driver", identificador do punto de E/S e frecuencia de monitorización.
b. Descargar unha táboa de E/S da memoria da máquina virtual.

3. *Carga, descarga e execución de modelos* no subsistema de aplicación da máquina virtual, nesta categoría agrúpanse os servicios que permiten:
   a. Cargar un modelo no subsistema de aplicación da máquina virtual para a súa posterior execución.
   b. Descargar un modelo do subsistema de aplicación da máquina virtual.
   c. Iniciar a execución dun modelo.
   d. Deter a execución dun modelo.
   e. Continuar a execución dun modelo. Este servicio utilízase na opción de execución paso a paso do modelo durante a súa depuración.

   A máquina virtual permite ter varios modelos almacenados simultaneamente na súa memoria. Antes de iniciar a execución dun modelo é preciso cargalo no módulo do subsistema de aplicación da máquina virtual no que vaia a executarse. Aínda que na versión actual da máquina virtual o subsistema de aplicación unicamente conteña un intérprete de modelos Grafcet, e este só acepte a carga dun único modelo, as mensaxes que implementan estes servicios foron definidas para permitir en futuras versións a utilización de múltiples interpretes e múltiples modelos.

   A execución dun modelo pode facerse nun de dous modos: depuración ou explotación. No modo de depuración a máquina virtual envía, despois de cada evolución do modelo a información sobre a situación e os valores das súas variábeis. Despois detense a súa evolución ata que non se reciba unha mensaxe de continuación ou de detención da execución. Este modo permite tanto a visualización remota do estado como o control paso a paso da evolución dos modelos. Combinando este modo coa simulación remota de E/S (§7.2.3) pode simularse o proceso e verificar o comportamento dos modelos antes da posta en funcionamento do sistema de control.

4. *Xestión de configuracións*, nesta categoría agrúpanse os servicios que permiten:
   a. Engadir unha nova configuración (§7.1.1.4) nun módulo da máquina virtual.
   b. Eliminar unha configuración dun módulo da máquina virtual.
   c. Activar unha configuración nun módulo da máquina virtual.
   d. Consultar a configuración activa nun módulo da máquina virtual.
   e. Consultar as configuracións dispoñíbeis nun módulo da máquina virtual e os tipos e módulos que as forman.
   f. Activar un módulo da configuración dun módulo da máquina virtual.
   g. Desactivar un módulo da configuración dun módulo da máquina virtual.

5. *Consulta da estructura de módulos* da máquina virtual, nesta categoría agrúpanse os servicios que permiten:
   a. Consultar a estructura de módulos da máquina virtual.
   b. Consultar os módulos dispoñíbeis nun almacén de módulos (§7.1.1.3) da máquina virtual.

6. *Consulta da información de configuración* (§7.2.1.2) dos "drivers" de E/S.
7. *Desconexión e finalización* da execución da máquina virtual. Nesta categoría agrúpanse os servicios que permiten:
   a. Desconectar ao cliente da máquina virtual, deixando a conexión libre para que un novo cliente poda acceder aos servicios remotos.
   b. Apagar a máquina virtual, detendo a súa execución e eliminando os recursos que ocupe na memoria.

Os códigos, patróns de comunicación, mensaxes e parámetros utilizados en cada servicio poden consultarse no Anexo F.

7.4.4. Modelado do acceso remoto á máquina virtual

O diagrama da Figura 7.35 mostra as clases definidas para implementar a comunicación remota entre un cliente e a máquina virtual. A clase *VMCommand* representa as mensaxes intercambiadas nesta comunicación e os seus atributos correspóndense cos campos indicados en (§7.4.1). A clase abstracta *IVMClient* declara a interface a través da que un cliente accede aos servicios remotos da máquina virtual. A declaración desta interface é a seguinte:

```
2669. struct IVMClient
2670. {
2671.  // carga e descarga de DLLs
2672.  virtual bool loadDLL(const string& name, const string& path) = 0;
2673.  virtual bool unloadDLL(const string& name) = 0;
2674.  virtual bool unloadAllDLLs() = 0;
2675.  // carga e descarga de táboas de E/S
2676.  virtual bool loadSystemIO(const string& path) = 0;
2677.  virtual bool unloadSystemIO() = 0;
2678.  // carga, descarga e execución de modelos
2679.  virtual bool loadModel(const string& module, const string& model) = 0;
2680.  virtual bool unloadModel(const string& module, const string& model) = 0;
2681.  virtual bool playModel(const string& module,
2682.                         const string& model,
2683.                         unsigned opts) = 0;
2684.  virtual bool nextModelEvolution(const string& module,
2685.                         const string& model) = 0;
2686.  virtual bool stopModel(const string& module, const string& model) = 0;
2687.  // xestión de configuracións
2688.  virtual bool addConfiguration(const string& module,
2689.                         const ModuleConfiguration& conf) = 0;
2690.  virtual bool removeConfiguration(const string& module,
2691.                         const string& name) = 0;
2692.  virtual bool setCurrentConfiguration(const string& module,
2693.                         const string& name) = 0;
2694.  virtual bool getCurrentConfiguration(const string& module,
2695.                         ModuleConfiguration& conf) = 0;
2696.  virtual bool getConfigurationInfo(const string& module,
2697.                         ModuleConfSeq& conf) = 0;
2698.  virtual bool activate(const string& module,
2699.                         const string& conf,
2700.                         const string& type,
2701.                         const string& name) = 0;
2702.  virtual bool deactivate(const string& module,
2703.                         const string& conf,
2704.                         const string& type,
2705.                         const string& name) = 0;
2706.  // consulta da estructura de módulos da máquina virtual
2707.  virtual bool getModuleInfo(const string& module, ModuleInfo& info) = 0;
2708.  virtual bool getVMInfo(VMInfo& info) = 0;
```

```
2709.  // consulta da información de configuración dos "drivers" de E/S
2710.  virtual bool getDeviceInfo(const string& device, DeviceInfo& ddinfo) = 0;
2711.  // finalización da execución da máquina virtual
2712.  virtual bool shutdown() = 0;
2713. };
```

Os métodos desta interface correspóndense cos servicios da máquina virtual explicados no apartado anterior. O único servicio para o que non se define o correspondente método é o de desconexión, xa que se supón que esta se realiza cando o cliente é destruído, e polo tanto será implementado no destructor das clases derivadas de *IVMClient*.

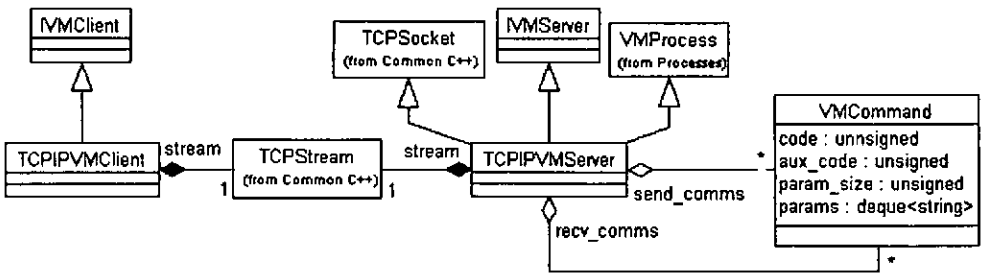


Figura 7.35. Diagrama das clases utilizadas para modelar o acceso dun cliente aos servicios remotos da máquina virtual.

A xestión do envío e recepción de mensaxes na máquina virtual é realizada por un proceso que implemente a interface definida pola clase abstracta *IVMServer*. O código da declaración desta interface é o seguinte:

```
2714. struct IVMServer
2715. {
2716.  // control de estado
2717.  virtual void Start() = 0;
2718.  virtual void Suspend() = 0;
2719.  virtual void Resume() = 0;
2720.  virtual void Finish() = 0;
2721.  virtual bool isSuspended() = 0;
2722.  virtual bool isRunning() = 0;
2723.  // recepción de mensaxes
2724.  virtual void onCommand(pfn fun, IAsyncClbkArg* fun_arg = NULL) = 0;
2725.  virtual void getCommand(SFCVMCommand*& command) = 0;
2726.  // envío de mensaxes
2727.  virtual bool ok(unsigned code = 0) = 0;
2728.  virtual bool ok(const deque<string>& params, unsigned code = 1) = 0;
2729.  virtual bool fail(unsigned err = 0) = 0;
2730. };
```

Ademais dos métodos que permiten controlar o estado do proceso (liñas 2717-2722), esta interface declara métodos para solicitar unha notificación asíncrona cando se reciba unha mensaxe —método *onCommand* (liña 2724)—, para recuperar as mensaxes recibidas —método *getCommand* (liña 2725)— e para enviar as mensaxes de resposta —métodos *ok* (liñas 2727 e 2728) e *fail* (liña 2729)—.

7.4.4.1. Implementación do acceso remoto en redes TCP/IP

O mecanismo de acceso remoto á máquina virtual en redes TCP/IP implementouse nas clases *TCPIPVMClient* e *TCPIPVMServer*, derivadas das interfaces *IVMClient* e *IVMServer* respectivamente. A comunicación entre as instancias destas clases realízase mediante instancias da clase *TCPStream*, pertencente á libraría Common C++ (§1.3.2). Esta clase representa unha

conexión entre un cliente e un servidor TCP, e permite o envío e recepción de datos mediante os operadores << e >>, do mesmo xeito que se fai nos "streams" C++. As instancias da clase *TCPIPVMServer* son procesos da máquina virtual que proporcionan un servidor TCP que implementa a xestión do envío, recepción e almacenamento —agregacións *recv_comms* e *send_comms*— das mensaxes intercambiadas pola máquina virtual cos seus clientes. A clase *TCPSocket*, tamén pertencente á libraría Common C++, proporciona un servidor TCP básico a partir do cal se implementa o resto da funcionalidade da clase *TCPIPVMServer*.

O diagrama da Figura 7.37 mostra a secuencia de mensaxes entre as instancias que implementan o mecanismo de acceso remoto aos servicios da máquina virtual para realizar a carga dunha DLL. A máquina virtual configura o servidor TCP para recibir unha notificación cada vez que se reciba unha orde e despois inicia a súa execución invocando o seu método *Start* (liña 2717). A solicitude do servicio remoto é iniciada coa chamada ao método *loadDll* (liña 2672) na instancia da clase *TCPIPVMClient*. A implementación deste método crea a orde a enviar á máquina virtual, envíaa a través do "stream" TCP e agarda a mensaxe de resposta. Cando a orde é recibida no servidor TCP, almacénase na cola de ordes recibidas —agregación *recv_comms*— e unha notificación é enviada á máquina virtual. Esta recupera a orde recibida chamando ao método *getCommand* (liña 2725), a interpreta e executa o servicio solicitado. En función do resultado da execución a máquina virtual chama aos métodos *ok* (liñas 2727 e 2728) ou *fail* (liña 2729) no servidor para enviar unha mensaxe de resposta. Estes métodos crean a mensaxe correspondente, almacénana na cola de mensaxes de saída —agregación *send_comms*— e sinálanlle ao servidor a presencia dunha nova mensaxe. Este recupera a mensaxe e envíaa ao cliente a través do "stream" TCP.

7.4.4.2. Implementación do acceso simultáneo a múltiples máquinas virtuais

Para facilitar o desenvolvemento de aplicacións distribuídas, definiuse un conxunto de clases que permiten xestionar o acceso simultáneo a múltiples máquinas virtuais dende un cliente. O diagrama da Figura 7.36 mostra as clases utilizadas para modelar esta funcionalidade e implementala utilizando múltiples "threads" e o protocolo de comunicacións TCP/IP.

A xestión das múltiples conexións que un cliente pode establecer simultaneamente cunha máquina virtual son manexadas nunha secuencia de execución independente que implemente a interface *IVMConnectionMgr*, que proporciona métodos para iniciar e finalizar conexións coas máquinas virtuais e para acceder aos servicios que estas proporcionan:

```
2731. struct IVMConnectionMgr
2732. {
2733.  // conexións
2734.  virtual bool connect(IVMachineInfo* vm) = 0;
2735.  virtual bool disconnect(IVMachineInfo* vm) = 0;
2736.  // carga e descarga de DLLs
2737.  virtual bool loadDll(IVMachineInfo* vm,
2738.                const string& name,
2739.                const string& path) = 0;
2740.  virtual bool unloadDll(IVMachineInfo* vm, const string& name) = 0;
2741.  virtual bool unloadAllDlls(IVMachineInfo* vm) = 0;
2742. // xestión de configuracións
2743.  virtual bool addConf(IVMachineInfo* vm,
2744.                const string& part,
2745.                ModuleConfiguration* conf) = 0;
2746.  virtual bool removeConf(IVMachineInfo* vm,
2747.                const string& part,
2748.                const string& conf) = 0;
```

```
2749.  virtual bool selectConf(IVMachineInfo* vm,
2750.                const string& part,
2751.                const string& conf) = 0;
2752.  // xestión de módulos
2753.  virtual bool addModule(IVMachineInfo* vm,
2754.                const string& part,
2755.                const string& conf,
2756.                const string& type,
2757.                const string& module) = 0;
2758.  virtual bool removeModule(IVMachineInfo* vm,
2759.                const string& part,
2760.                const string& conf,
2761.                const string& type,
2762.                const string& module) = 0;
2763.  // consulta da estructura da máquina virtual
2764.  virtual bool getVMInfo(IVMachineInfo* vm) = 0;
2765.  // finalización da execución da máquina virtual
2766.  virtual bool shutdown(IVMachineInfo* vm) = 0;
2767. };
```

Como pode comprobarse, os métodos desta interface coinciden cos da interface *IVMClient*[75] coa diferencia de que teñen un parámetro máis para identificar a máquina virtual á que van dirixidas as solicitudes de servicio. Cando un cliente solicita un servicio, pasa como parámetro unha instancia que implemente a interface *IVMachineInfo* e que conteña, como mínimo, o identificador alfanumérico da máquina virtual, o enderezo numérico utilizado para establecer unha conexión con ela, e un apuntador ao cliente que realiza a solicitude[76]. As clases derivadas poden incluír outras informacións específicas do sistema de comunicacións utilizado.

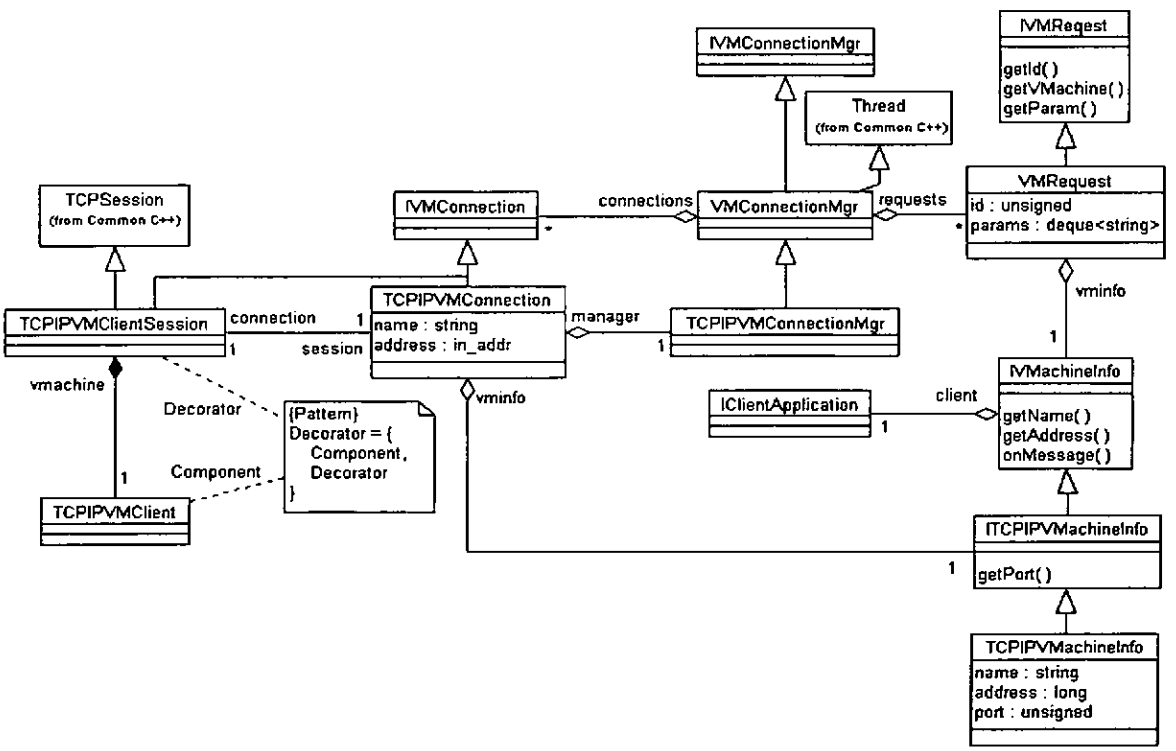


Figura 7.36. Clases utilizadas para modelar o acceso dun cliente a múltiples máquinas virtuais en redes TCP/IP.

[75] A principal diferencia é que nesta interface os métodos de *IVMClient* utilizados para consultar a información sobre os módulos, configuracións e "drivers" de E/S da máquina virtual son substituidos polo método *getVMInfo*.

[76] Representado na Figura 7.36 mediante a interface *IClientApplication*, que se deixou sen especificar intencionadamente.

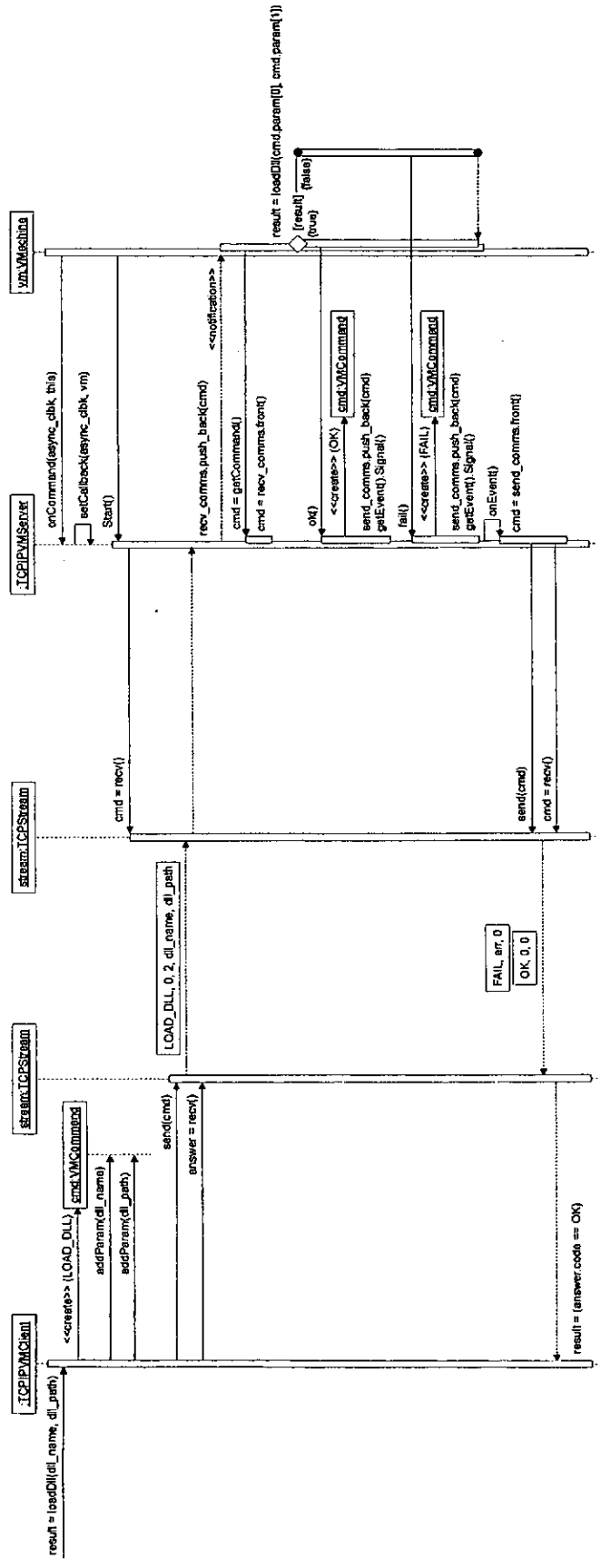


Figura 7.37. Secuencia de mensaxes intercambiadas no acceso aos servicios remotos da máquina virtual nunha rede TCP/IP.

As instancias da clase *VMConnectionMgr* son "threads" que proporcionan unha implementación por defecto da interface *IVMConnectionMgr*, utilizando para elo a funcionalidade derivada da clase *Thread* pertencente á libraría Common C++. Cada instancia almacena información sobre as solicitudes de servicio pendentes —agregación *requests*— e as conexións coas máquinas virtuais abertas —agregación *connections*—. As solicitudes de servicio son representadas mediante instancias da clase *VMRequest*, derivada da interface *IVMRequest*, que almacenan o código e os parámetros do servicio solicitado, así como un apuntador á información da máquina virtual recibida do cliente cando este solicita o servicio —agregación *vminfo*—. No referente ás conexións, son representadas mediante instancias de clases derivadas da interface *IVMConnection*, cuxa declaración é a seguinte:

```
2768. struct IVMConnection
2769. {
2770.   // conexións
2771.   virtual bool connect() = 0;
2772.   virtual bool disconnect() = 0;
2773.   // carga e descarga de DLLs
2774.   virtual bool loadDll(const string& name, const string& path) = 0;
2775.   virtual bool unloadDll(const string& name) = 0;
2776.   virtual bool unloadAllDlls() = 0;
2777.   // xestión de configuracións
2778.   virtual bool addConf(const string& part, ModuleConfiguration* conf) = 0;
2779.   virtual bool removeConf(const string& part, const string& conf) = 0;
2780.   virtual bool selectConf(const string& part, const string& conf) = 0;
2781.   // xestión de módulos
2782.   virtual bool addModule(const string& part,
2783.                          const string& conf,
2784.                          const string& type,
2785.                          const string& module) = 0;
2786.   virtual bool removeModule(const string& part,
2787.                          const string& conf,
2788.                          const string& type,
2789.                          const string& module) = 0;
2790.   // consulta da estructura da máquina virtual
2791.   virtual bool getVMInfo(VMArchitectureInfo* pInfo) = 0;
2792.   // finalización da execución da máquina virtual
2793.   virtual bool shutdown() = 0;
2794. };
```

A declaración desta interface é semellante á da interface *IVMClient*. Isto permite que as instancias das clases derivadas de *IVMConnection* poidan ser utilizadas como decoradores [67] das instancias de clases derivadas de *IVMClient* e, polo tanto, implementar de forma transparente funcións adicionais no acceso aos servicios da máquina virtual. As clases *TCPIPVMConnectionMgr*, *TCPIPVMachineInfo* e *TCPIPVMConnection* (Figura 7.36) proporcionan unha implementación das interfaces *IVMConnectionMgr*, *IVMachineInfo* e *IVMConnection* para redes que utilicen o protocolo de comunicacións TCP/IP. Nesta implementación cada conexión é manexada nunha sesión independente representada pola clase *TCPIPVMClientSession*. Esta clase implementa a interface *IVMConnection* utilizando a funcionalidade derivada da clase *TCPSession*, pertencente á libraría Common C++. Cada instancia da clase *TCPIPVMClientSession* é, polo tanto, un "thread" que xestiona unha conexión coa máquina virtual servindo de decorador dunha instancia da clase *TCPIPVMClient* —agregación *vmachine*—. Esta instancia é a que proporciona o acceso remoto aos servicios da máquina virtual tal e como se explicou no apartado anterior (Figura 7.35).

A Figura 7.38 e a Figura 7.39 mostran exemplos das secuencias de mensaxes intercambiadas entre as instancias das clases anteriores para o establecemento dunha conexión cunha máquina virtual remota e a solicitude de carga dunha DLL, respectivamente. Nótese que a finalización

do servicio solicitado se lle indica ao cliente invocando o método *onMessage* (Figura 7.36) da instancia da clase derivada de *IVMachineInfo* recibida como parámetro da solicitude. Esta instancia mantén un apuntador ao cliente que solicitou o servicio, que será notificado da finalización do mesmo. A implementación actual da clase *VMConnectionMgr* ten a limitación de bloquearse durante o procesamento dunha solicitude ata que esta remate. En futuras versións poderían implementarse xestores que procesen múltiples solicitudes simultáneas, incorporando a cada conexión a capacidade de almacenar as solicitudes e notificarlle ao xestor a súa finalización.

7.5. O subsistema de aplicación

O subsistema de aplicación é a compoñente da arquitectura na que se cargan dinamicamente e se executan os modelos de usuario. Este subsistema está deseñado para ser utilizado con aplicacións que interaccionen co proceso mediante o intercambio de eventos. A versión actual contén unicamente un intérprete Grafcet pero proporciona a infraestructura "software" precisa para incluír en futuras versións outros tipos de intérpretes (p.e, RdPI, diagramas de estados, etc.) e aplicacións. Neste apartado explícanse os aspectos da interacción entre este subsistema e o proceso. A implementación do intérprete Grafcet explícase no Capítulo 8.

A interacción do subsistema de aplicación co proceso realízase mediante dous canais, un de entrada polo que o subsistema recibe os eventos detectados no proceso; e outro de saída polo que se envían os valores das saídas calculados pola aplicación. A Figura 7.40 e a Figura 7.41 mostran de maneira simplificada as secuencias de mensaxes intercambiadas entre os procesos da máquina virtual dende que se obtén o valor dunha entrada do proceso ata que o subsistema de aplicación procesa ese valor. Como pode verse esta interacción non é directa, senón que nela interveñen diferentes procesos do subsistema de E/S e do núcleo da máquina virtual.

Como se explica en (§7.2.1.3), os "drivers" de E/S poden configurarse tanto para monitorizar o valor dunha entrada a intervalos regulares (lectura síncrona), como para obter o valor da entrada cada vez que este cambie (lectura asíncrona). Nas dúas secuencias mostradas na Figura 7.40 e na Figura 7.41, o "driver" utilizado está configurado para obter o valor da entrada a intervalos regulares, mais a secuencia seria semellante se estivera configurado para obter un valor cada vez que se detectase un cambio na entrada.

Do mesmo xeito, os canais para o paso de mensaxes (§7.1.2.4.1) proporcionan mecanismos tanto para o envío dunha notificación asíncrona cada vez que se almacena unha nova mensaxe, como para a lectura con bloqueo das mensaxes que almacenen. Na secuencia da Figura 7.40 o canal de entrada do subsistema de aplicación envíalle unha notificación asíncrona ao subsistema cada vez que se recibe un novo evento, mentres que na secuencia da Figura 7.41, é o subsistema de aplicación quen obtén a intervalos regulares os eventos almacenados no canal de entrada. Nótese que a base de datos unicamente crea un evento cando o valor obtido é o dunha variábel booleana e, ademais, é diferente do último valor almacenado.

A combinación das características anteriores permite a utilización do subsistema de aplicación para executar tanto procesos dirixidos por eventos, activados cada vez que se reciba un novo evento no canal de entrada, como procesos cíclicos, que accedan aos eventos de entrada como parte do seu ciclo de execución. A arquitectura proposta ten dous inconvenientes que condicionan á súa aplicabilidade en procesos con requisitos temporais estrictos:

1. Os canais para o paso de mensaxes entre procesos non implementan ningún mecanismo que impida o desbordamento das colas de mensaxes. Isto implica que o funcionamento da máquina virtual pode colapsarse se é utilizada con procesos que xeren eventos a un ritmo

maior do que esta sexa capaz de procesar. Este problema pode reducirse en certa medida mediante a especificación de frecuencias de monitorización nas variábeis de entrada, como se explica en (§7.2.1.4.2).

2. A máquina virtual introduce unha latencia entre o instante no que un "driver" obtén un valor de entrada e o instante no que o subsistema de aplicación recibe o evento correspondente. O valor deste retardo é variábel e débese ao "overhead" introducido polo subsistema de E/S e o núcleo da máquina virtual no procesamento dos valores de entrada.

7.6. Conclusións

Neste capítulo describiuse a arquitectura e o funcionamento da máquina virtual que da soporte á execución dun interprete Grafcet para a interpretación dos modelos de usuario. A flexibilidade e portabilidade da arquitectura permite a súa utilización en diferentes sistemas e con múltiples dispositivos de E/S. Explicáronse as técnicas implementadas para utilizar a máquina virtual en ambientes de simulación distribuídos e os servicios remotos que proporciona para a xestión da súa configuración e o control do seu funcionamento. Tamén se describiron os servicios básicos proporcionados pola máquina virtual, como a xestión de eventos e temporizacións ou a consulta, almacenamento e actualización dos valores do proceso.

Aínda que a máquina virtual foi deseñada inicialmente coa intención de executar un intérprete Grafcet, a arquitectura resultante proporciona un ambiente cos servicios precisos para implementar e executar outras aplicacións dirixidas por eventos. Un dos aspectos que se describiron en detalle no capítulo foi como é realizada a interacción entre o subsistema de aplicación e o proceso, poñendo de relevo os diferentes tipos de aplicacións soportados e indicando as latencias introducidas pola máquina virtual e as consecuencias na súa aplicación en sistemas con restriccións temporais estrictas. Para reducir en parte este problema describiuse unha técnica implementada nos "drivers" de E/S baseada na monitorización de entradas.

Entre as melloras que poderían implementarse en futuras versións poden citarse as seguintes:

1. A implementación dun mecanismo que permita configurar a resposta do sistema en caso de producirse un desbordamento das colas de mensaxes almacenadas nos canais de comunicación.
2. O soporte á utilización dalgunha linguaxe de definición de "drivers" de E/S ou método semellante, que permita reaproveitar as especificacións utilizadas noutros sistemas.
3. A substitución das comunicacións baseadas no intercambio de mensaxes pola utilización dalgunha infraestructura para o soporte a distribución de obxectos como a proporcionada por CORBA [126].

Por último indicar que a versión actual da máquina virtual non inclúe un mecanismo específico para a comunicación directa entre máquinas virtuais que permita a execución distribuída de Grafcets. Nótese sen embargo que é posíbel implementar este mecanismo utilizando a mesma técnica empregada para a simulación de E/S, intercambiando información entre as máquinas virtuais a través de "drivers" de E/S implementados a tal efecto.

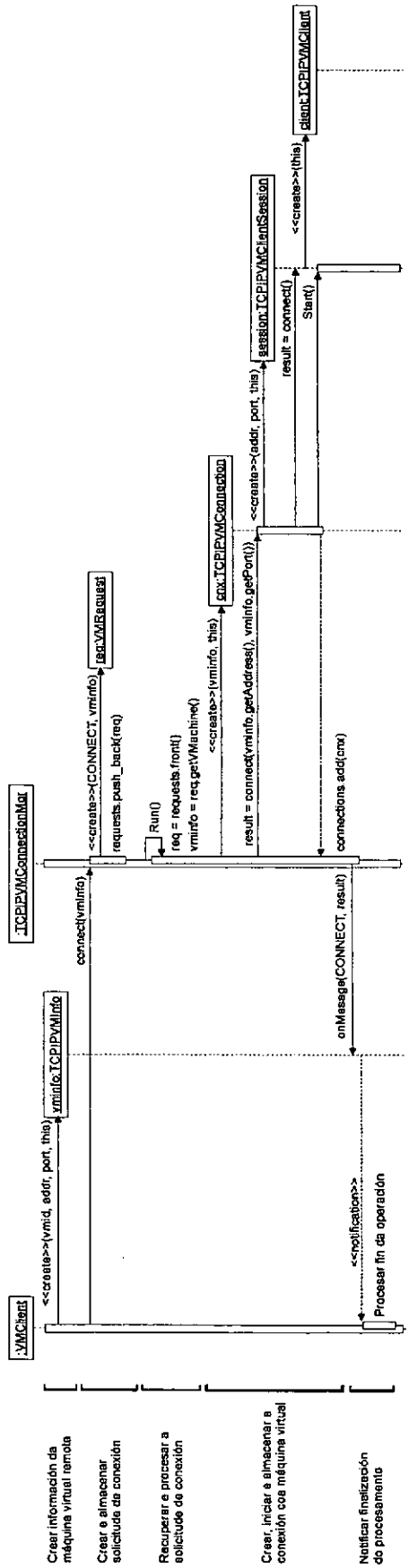


Figura 7.38. Secuencia das mensaxes intercambiadas para a creación dunha conexión coa máquina virtual.

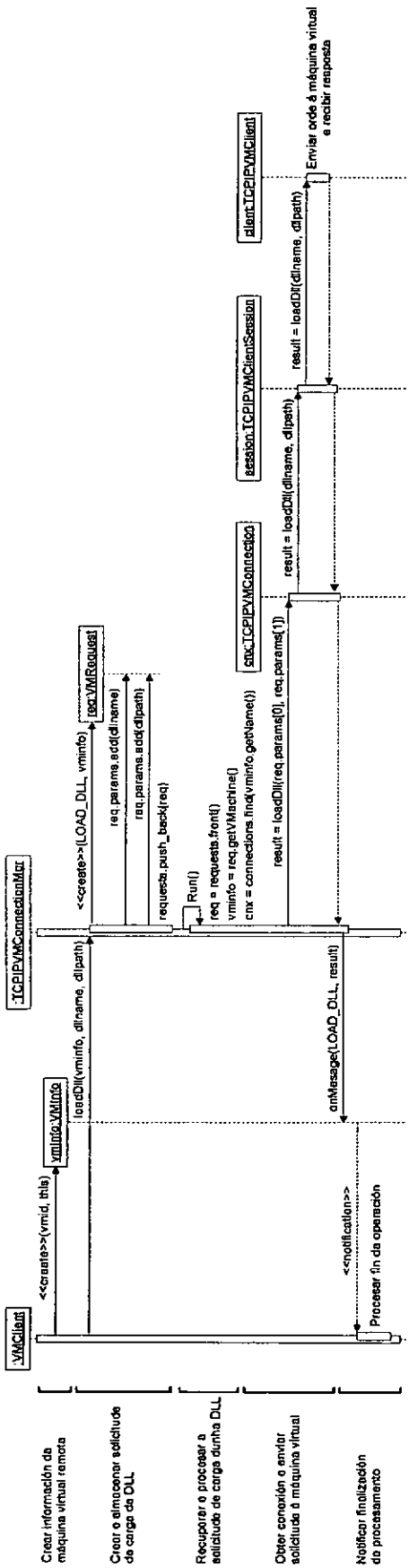


Figura 7.39. Secuencia das mensaxes intercambiadas para a solicitude de carga dunha DLL na máquina virtual.

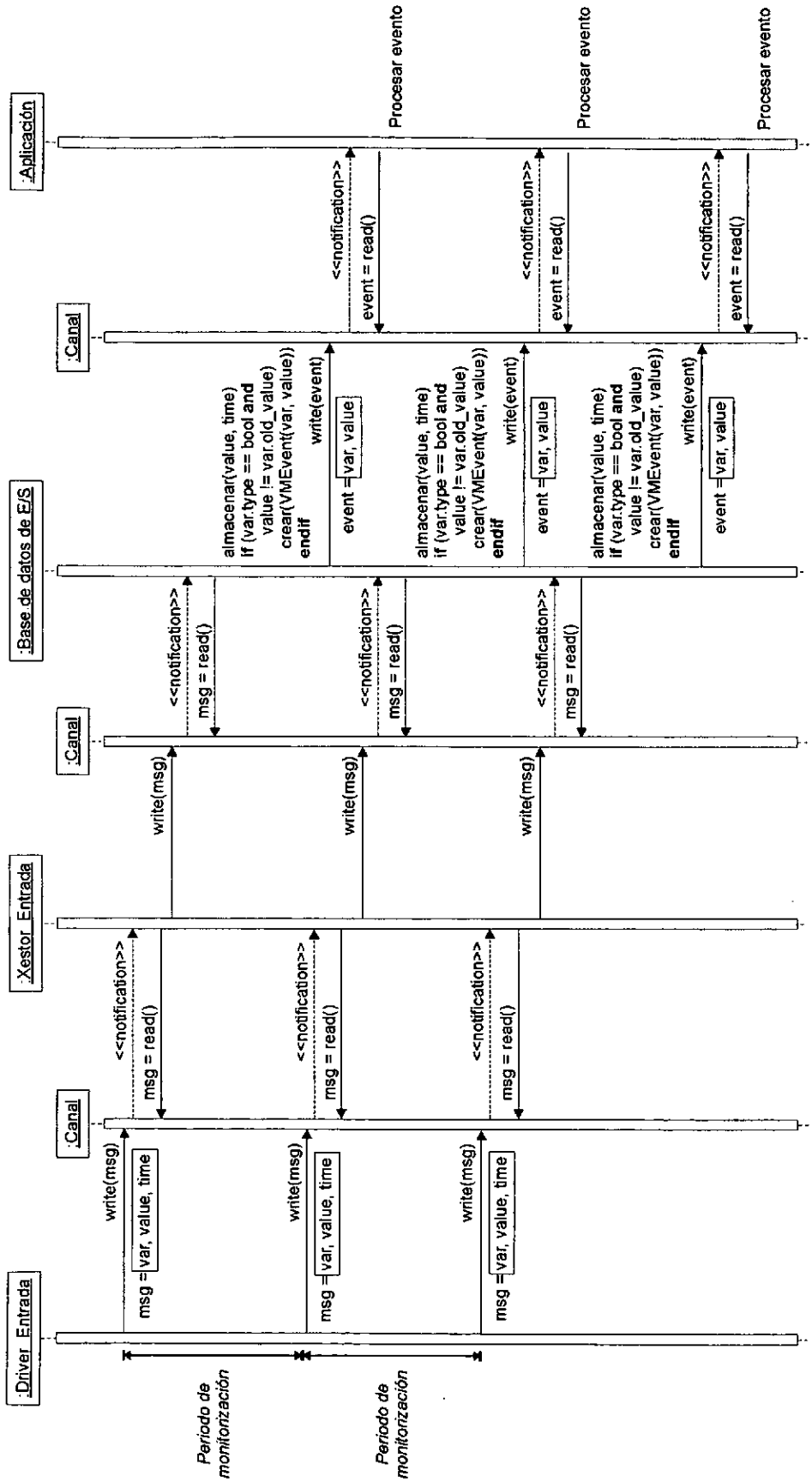


Figura 7.40. Secuencia das mensaxes intercambiadas para o procesamento dun evento detectado no proceso nunha aplicación dirixida por eventos.

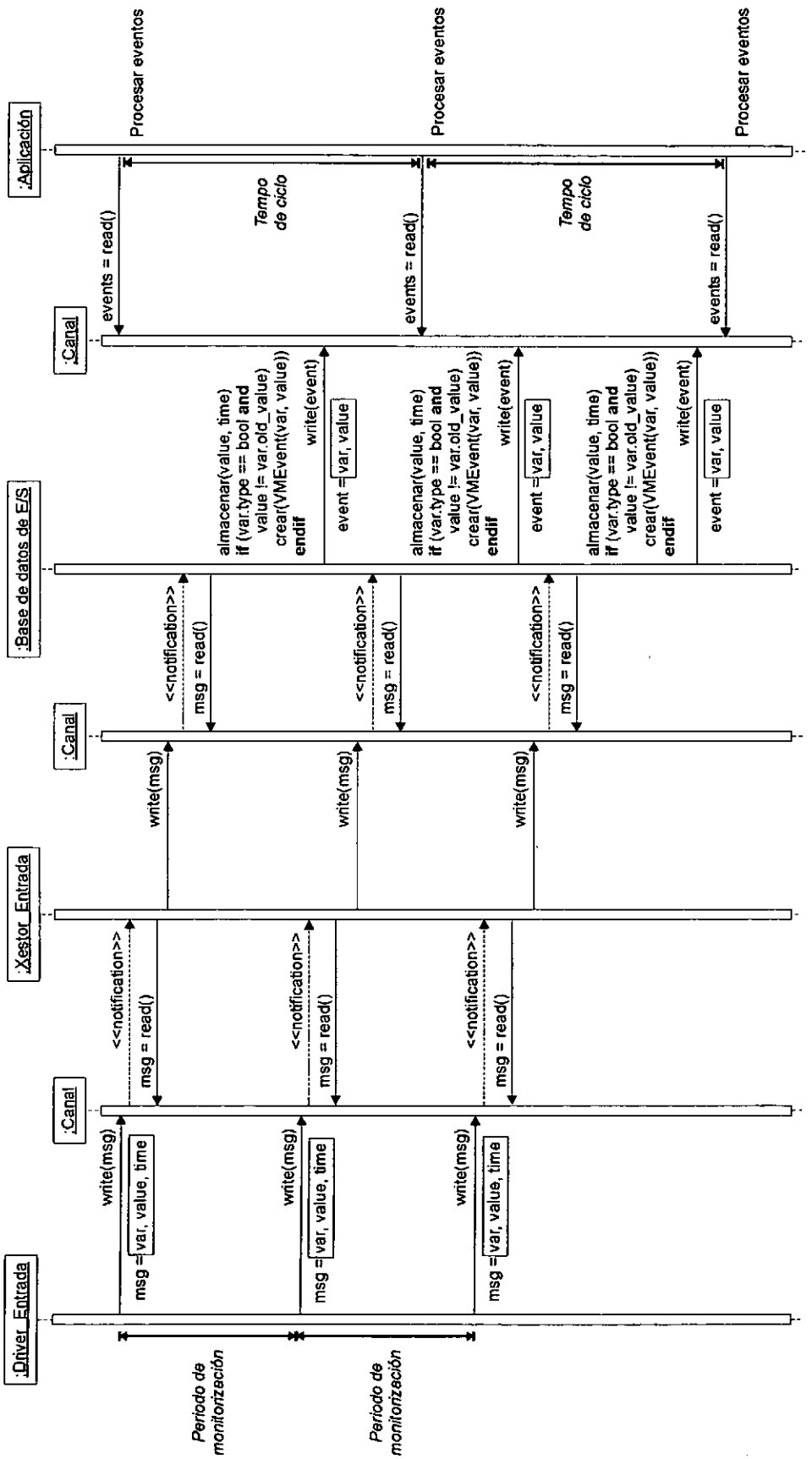


Figura 7.41. Secuencia das mensaxes intercambiadas para o procesamento dun evento detectado no proceso nunha aplicación cíclica.

Capítulo 8. O intérprete de modelos Grafcet

O intérprete Grafcet é o proceso da máquina virtual que se encarga da interpretación dos modelos Grafcet cargados na súa memoria. A interacción do intérprete co proceso físico realízase mediante o intercambio de eventos utilizando os canais de comunicación do subsistema de aplicación da máquina virtual (§7.5). O funcionamento do intérprete está composto polas seguintes fases que se executan ciclicamente:

1. A obtención dos eventos de entrada a considerar na seguinte evolución do modelo.
2. O cálculo das temporizacións.
3. A evolución do modelo Grafcet.
4. A execución das accións activas.
5. A actualización dos valores das saídas.

Dende o punto de vista do manexo dos eventos o intérprete ten unha semántica RTC, na que non se teñen en conta os novos eventos que se produzan durante unha evolución do modelo ata que esta remate. Esta semántica xunto coa comunicación mediante o paso de mensaxes garante que non se produzan conflictos ("race conditions") durante a execución do intérprete.

O ritmo ao que se procesan os eventos do proceso depende do tempo de ciclo do intérprete, que é variábel e dependente do modelo Grafcet interpretado. Isto implica que teoricamente só poda garantirse a reactividade do intérprete en procesos cun ritmo de cambio inferior ao do tempo de ciclo máximo. En (§3.3.2.4) descríbese un algoritmo de interpretación Grafcet que garante a reactividade supoñendo o caso ideal no que este tempo é instantáneo comparado co tempo de resposta requirido polo proceso (hipótese de causalidade nula). Sen embargo na práctica esta hipótese ideal reláxase, e considérase que o interprete garante a reactividade se o seu tempo de ciclo é inferior ao tempo de resposta máximo permitido polo proceso (esta aproximación é semellante á utilizada nos PLCs). En (§7.2.1.4.2) explícase unha técnica que utiliza os tempos de monitorización das variábeis de entrada para reducir en parte este problema.

O intérprete Grafcet foi deseñado aproveitando o mecanismo de substitución dinámica de módulos (§7.1.1.5) para configurar diferentes aspectos da súa arquitectura, como a obtención dos eventos do proceso ou o tipo de semántica (§3.3.2.2) a utilizar na interpretación dos modelos. O resultado é un intérprete cunha arquitectura flexíbel que pode ser utilizado en diferentes aplicacións, como por exemplo: a simulación e control de sistemas dirixidos por

eventos, a análise de extensións ao Grafcet que afecten aos algoritmos de interpretación, a emulación de PLCs, etc.

A organización deste capítulo é a seguinte: a arquitectura do intérprete é presentada no apartado (§8.1) e a súa implementación en (§8.2); os módulos e parámetros que permiten configurar o intérprete son detallados en (§8.3); o apartado (§8.4) describe os detalles do seu funcionamento; a información utilizada durante a interpretación dun modelo Grafcet detállase en (§8.5); a implementación da política de execución e as técnicas utilizadas para optimizar a avaliación de condicións e a execución de accións explícanse en (§8.6); finalmente, no apartado (§8.7) resúmense as conclusións do capítulo.

8.1. Arquitectura do intérprete

O intérprete Grafcet ten unha arquitectura lóxica composta polos módulos mostrados na Figura 8.1. Os rectángulos cun contorno máis fino correspóndense con módulos que poden ser substituídos dinamicamente para modificar a configuración do intérprete. Esta arquitectura basease na proposta en [114] para a interpretación do Grafcet en sistemas síncronos, e nela poden distinguirse catro compoñentes principais:

1. *O xestor de eventos*, que se encarga do manexo dos eventos recibidos polo intérprete a través do canal de entrada do subsistema de aplicación. A configuración e o funcionamento do xestor de eventos poden modificarse mediante dous módulos substituíbeis que definen a política de acceso (§8.3.1.1) e a de reacción (§8.3.1.2).
2. *O algoritmo de interpretación*, que é o módulo encargado de interpretar os modelos Grafcet. Este algoritmo pode ser substituído dinamicamente, de xeito que un mesmo intérprete poda utilizar diferentes algoritmos. Ademais cada algoritmo pode configurarse mediante dous módulos substituíbeis que definen a política de evolución dos modelos (§8.3.1.4) e a de execución do seu código (§8.3.1.5).
3. *A interpretación Grafcet* ou xogo Grafcet, que é o módulo que contén a información estructural do modelo, o código de accións e condicións e a información asociada ao estado da súa evolución.
4. *O xestor de saída*, que xestiona o envío de mensaxes co valor das saídas modificadas durante a evolución do modelo ao canal de saída do subsistema de aplicación.

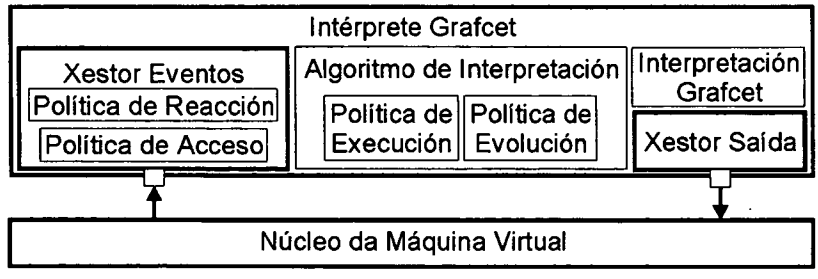


Figura 8.1. Arquitectura de módulos do intérprete Grafcet.

8.2. Implementación da arquitectura do intérprete

O intérprete é representado mediante a clase *SingleGrafcetPlayer*, cuxas relacións de especialización e agregación son mostradas no diagrama da Figura 8.2. Como se mostra neste diagrama, as instancias desta clase son procesos que implementan a interface *IGrafcetPlayer* e que inclúen os mecanismos para o almacenamento de módulos (§7.1.1.3) e configuracións

(§7.1.1.4), derivándoos respectivamente das clases *ModuleStore* e *ConfigurationStore*. A súa estructura se forma mediante a agregación de módulos, utilizando o soporte derivado da clase *StructuredModule*. O código da declaración da interface *IGrafcetPlayer* é o seguinte:

```
2795. struct IGrafcetPlayer
2796. {
2797.    // control do estado do intérprete
2798.    virtual void Start() = 0;
2799.    virtual void Suspend() = 0;
2800.    virtual void Resume() = 0;
2801.    virtual void Finish() = 0;
2802.    virtual bool isSuspended() = 0;
2803.    virtual bool isRunning() = 0;
2804.    // control da execución do modelo Grafcet
2805.    virtual bool loadModel(RTModel* model) = 0;
2806.    virtual bool unloadModel(const string& id) = 0;
2807.    virtual bool playModel(const string& model, // modo continuo
2808.                           TimeScale ev,
2809.                           TimeScale st,
2810.                           bool act,
2811.                           bool retrig,
2812.                           IVMServices* vm) = 0;
2813.    virtual bool playModel(const string& model, // modo paso a paso
2814.                           TimeScale ev,
2815.                           TimeScale st,
2816.                           bool act,
2817.                           bool retrig,
2818.                           IVMServices* vm,
2819.                           pfn callback,
2820.                           IAsyncClbkArg* parg = NULL) = 0;
2821.    virtual bool nextModelEvolution(const string& model) = 0;
2822.    virtual bool stopModel(const string& model) = 0;
2823.    virtual bool abortModel(const string& model, unsigned err_code = 0) = 0;
2824. };
```

Os métodos declarados nas liñas 2798-2803 son comúns aos declarados na interface *IVMProcess* (§7.1.2.1) para o control e consulta do estado de execución dun proceso. A carga e descarga de modelos Grafcet na memoria do intérprete realízase mediante os métodos *loadModel* (liña 2805) e *unloadModel* (liña 2806). Os modelos Grafcet estarán representados utilizando o formato xerado polo compilador Grafcet (§6.4). Os demais métodos son os que controlan a execución do modelo. O método *playModel*, que ten dúas versións, é utilizado para iniciar a execución do modelo, unha versión o executa de forma continua (liña 2807) e a outra (liña 2813) o executa no modo paso a paso (ciclo a ciclo). Esta segunda versión envía unha notificación asíncrona (§7.1.2.4.2) á máquina virtual cada vez que remata unha evolución do modelo e agarda a que se invoque o método *nextModelEvolution* (liña 2821) para iniciar a seguinte. Os atributos do método *playModel* permiten configurar a interpretación do modelo segundo se explica en (§8.3.2). Os métodos *stopModel* (liña 2822) e *abortModel* (liña 2823) serven para deter a execución do modelo, o primeiro remata a evolución actual antes de detelo e o segundo deteno inmediatamente sen rematar a evolución actual.

No que se refire as relacións de agregación definidas para representar a estructura do intérprete Grafcet (Figura 8.2), as agregacións *input* e *output* referencian aos xestores de eventos e saídas respectivamente, que son modelados mediante as clases *GrafcetPlayerInput* e *GrafcetPlayerOutput*. A agregación *algorithm* referencia ao algoritmo de interpretación, que é unha instancia dunha clase que implemente a interface *IGrafcetPlayerAlgorithm*. Finalmente, a interpretación do modelo Grafcet é representada coa clase *GrafcetGame*, que é referenciada mediante a agregación *game*. O intérprete accede aos servicios proporcionados polo núcleo da

máquina virtual mantendo unha referencia —agregación *vmachine*— a unha instancia da clase *IVMServices* (§7.3.3) que recibe como argumento do método *playModel* (liña 2807).

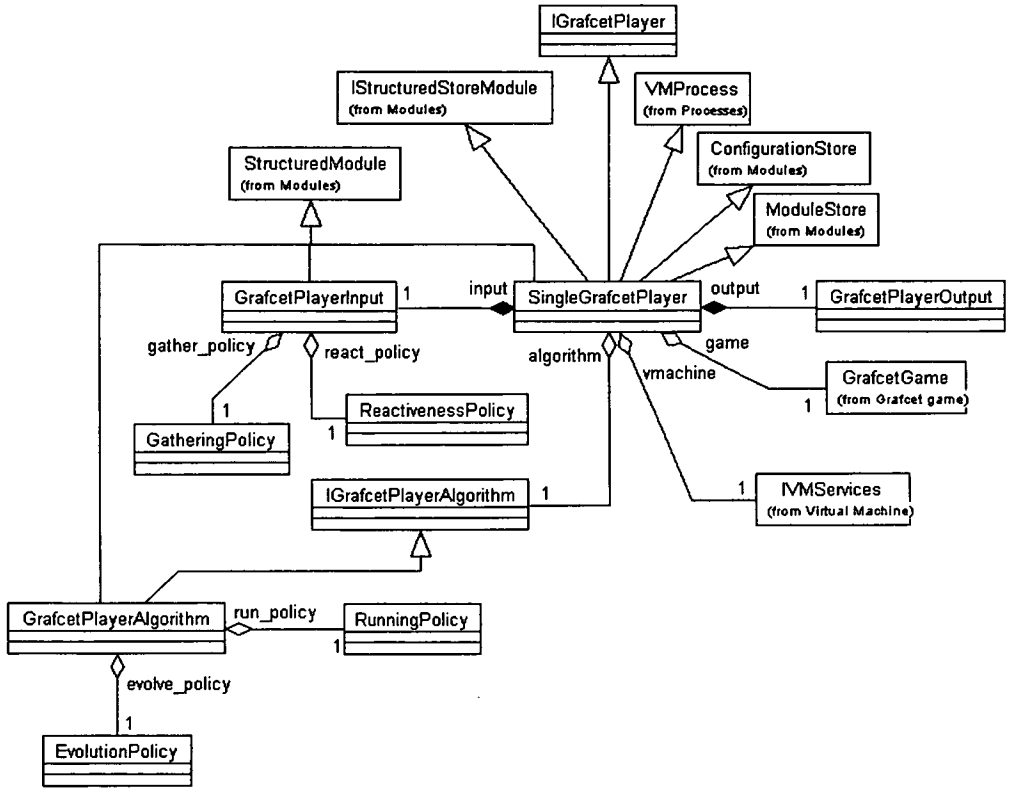


Figura 8.2. Diagrama das clases utilizadas para implementar a estructura do intérprete Grafcet.

A estructura do intérprete complétase coas agregacións definidas nas clases *GrafcetPlayerInput* e *GrafcetPlayerAlgorithm* para referenciar aos módulos substituíbeis que permiten modificar a súa configuración. No caso da clase *GrafcetPlayerInput*, a agregación *gather_policy* referencia á política de acceso aos eventos que é modelada mediante a clase *GatheringPolicy*, e a agregación *react_policy* referencia á política de reacción que é modelada mediante a clase *ReactivenessPolicy*.

No referente á clase *GrafcetPlayerAlgorithm*, derivada de *IGrafcetPlayerAlgorithm*, tamén define dúas relacións de agregación: *evolve_policy* que referencia á política de evolución de modelos, representada mediante a clase *EvolutionPolicy*, e *run_policy* á política de execución do código dos modelos, representada mediante a clase *RunningPolicy*.

Nótese que a arquitectura mostrada na Figura 8.1 correspóndese cun intérprete que xestiona unha única interpretación dun único modelo. Utilizando a mesma arquitectura é posíbel implementar intérpretes que xestionen múltiples interpretacións dun ou varios modelos diferentes, simplemente substituíndo o algoritmo de interpretación e almacenando varios xogos simultaneamente.

8.3. Configuración do intérprete

A configuración da arquitectura e do funcionamento do intérprete realízase utilizando dúas técnicas diferentes: mediante o mecanismo de carga, descarga e substitución dinámica de módulos utilizado na implementación da máquina virtual (§7.1.1.5), e mediante parámetros que

dada unha configuración específica modifiquen certos aspectos do seu funcionamento. A continuación descríbense os detalles das dúas técnicas utilizadas.

8.3.1. Módulos substituíbeis no intérprete

Como pode verse na Figura 8.1 na arquitectura do intérprete poden substituírse ata cinco tipos distintos de módulos que modifican diferentes aspectos do seu comportamento:

1. *A política de acceso* aos eventos detectados no proceso.
2. *A política de reacción* diante dos eventos detectados no proceso.
3. *O algoritmo de interpretación.*
4. *A política de evolución* dos modelos.
5. *A política de execución* do código dos modelos.

A Figura 8.3 mostra o diagrama de clases dos diferentes módulos definidos na versión actual do intérprete para configurar a súa estructura. A continuación descríbese a funcionalidade e as posíbeis configuracións de cada un dos tipos de módulos indicados.

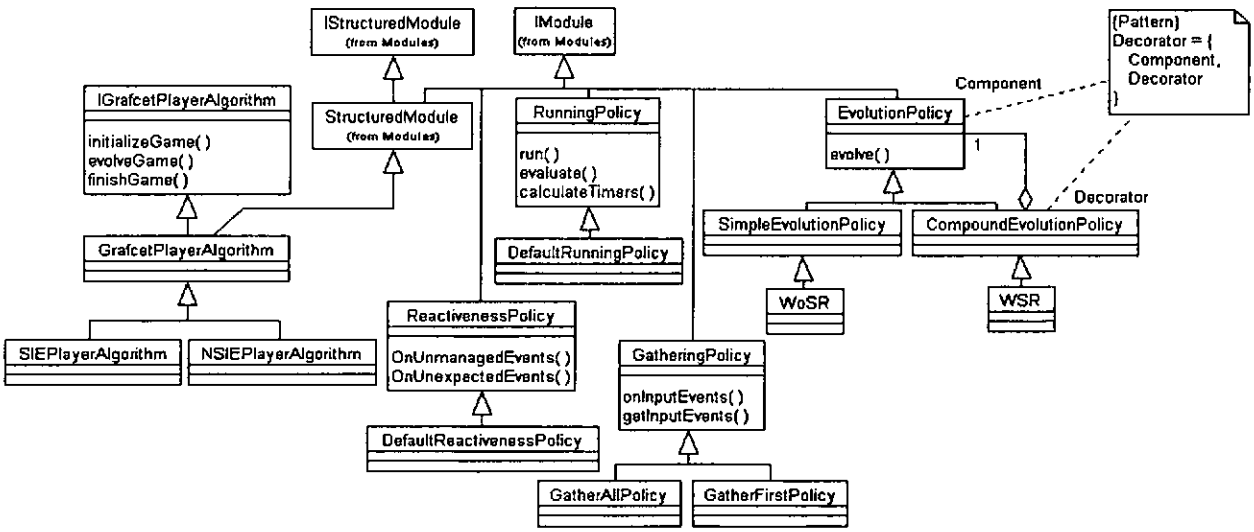


Figura 8.3. Diagrama de clases dos módulos utilizados para configurar o intérprete Grafcet.

8.3.1.1. A política de acceso aos eventos

Este módulo é utilizado para realizar o acceso aos eventos almacenados no canal de entrada do subsistema de aplicación no que se executa o intérprete. Como pode verse na Figura 8.3 a política de acceso está representada mediante a clase *GatheringPolicy*, que declara dous métodos virtuais a redefinir nas clases derivadas:

1. *onInputEvents*, invocado cando hai novos eventos de entrada para filtrar os que non sexan considerados válidos. A implementación por defecto deste método unicamente almacena os eventos recibidos sen realizar ningún filtrado.
2. *getInputEvents*, invocado polo xestor de eventos para seleccionar o(s) evento(s) de entrada a considerar na seguinte evolución do modelo. Dos eventos almacenados polo método *onInputEvents*, este método selecciona os que van ser utilizados na seguinte evolución. Na versión actual do intérprete hai dúas posibilidades, representadas na Figura 8.3 mediante as

clases *GatherAllPolicy* e *GatherFirstPolicy*, que devolven respectivamente todos os eventos almacenados e unicamente o primeiro dos eventos almacenados.

8.3.1.2. A política de reacción

A política de reacción, representada na Figura 8.3 mediante a clase *ReactivenessPolicy*, permite configurar o comportamento do intérprete ante algunha das circunstancias seguintes:

1. A ocorrencia de eventos de entrada non significativos. Considérese o modelo Grafcet da Figura 8.4 no que a situación actual é S = {2, 3, 7} e as variábeis de entrada do proceso son *a*, *b*, e *c*, todas booleanas e independentes entre si. O conxunto de posíbeis eventos de entrada do modelo será polo tanto E = {↑a, ↓a, ↑b, ↓b, ↑c, ↓c}. Sexa I ⊆ E o conxunto de eventos a considerar nun momento determinado para avaliar as condicións das transicións validadas e, se é o caso, iniciar unha nova evolución do modelo. Este conxunto será significativo (pode provocar o franqueamento dunha transición) unicamente se contén eventos que afecten ás variábeis das transicións validadas, neste exemplo {↑a, ↓a, ↑b, ↓b}. Os eventos que non pertenzan a este conxunto poden descartarse pois non modifican a situación do modelo[77].

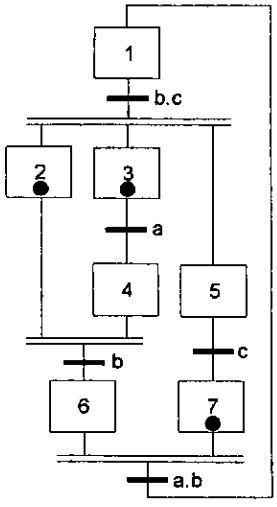


Figura 8.4. Grafcet utilizado como exemplo.

2. A ocorrencia dun evento de entrada durante unha evolución do modelo. Como se explica en (§3.3.2.4), a interpretación reactiva dun modelo Grafcet basease na hipótese ideal da causalidade nula, na que o tempo de evolución do modelo é instantáneo comparado co tempo de resposta esixido polo proceso. Isto implica que non existe a posibilidade de que se reciba un novo evento mentres se realiza unha evolución iniciada pola ocorrencia dun evento anterior. Sen embargo nunha aplicación real non é posíbel garantir esta hipótese e moitos sistemas funcionan proporcionando un tempo de resposta o suficientemente rápido, comparado co do proceso, para permitir o procesamento de todos os eventos de entrada cun retraso aceptábel.

A clase *ReactivenessPolicy* declara os métodos virtuais *onUnexpectedEvents* e *onUnmanagedEvents* que poden ser redefinidos nas clases derivadas para configurar a resposta nas circunstancias anteriores dependendo da utilización que se estea a facer do intérprete. Na

[77] Nótese que este é un exemplo simple, nun caso real habería que ter en conta tamén as condicións das temporizacións e das accións condicionais asociadas a etapas activas.

implementación por defecto, proporcionada pola clase *DefaultReactivenessPolicy*, todos os eventos son considerados como significativos e os eventos recibidos durante unha evolución do modelo son almacenados para o seu procesamento posterior.

8.3.1.3. O algoritmo de interpretación

A interface *IGrafcetPlayerAlgorithm* (Figura 8.3) declara os métodos que permiten iniciar, evoluir e finalizar a execución do algoritmo de interpretación de modelos Grafcet. A clase derivada *GrafcetPlayerAlgorithm*, que proporciona a implementación por defecto, é unha aplicación do patrón de deseño "template method" [67]. É dicir, esta clase proporciona a estructura xenérica do algoritmo de interpretación, que para realizar a evolución do modelo e para executar o código utiliza os métodos proporcionados polas políticas de evolución e execución respectivamente.

Como pode verse na Figura 8.3, a versión actual da máquina virtual proporciona dous algoritmos de interpretación diferentes, representados mediante as clases *SIEPlayerAlgorithm* e *NSIEPlayerAlgorithm*. A diferencia entre ambos está no momento no que teñen en conta os eventos do proceso recibidos durante unha evolución do modelo Grafcet. Na Figura 3.37 móstrase un exemplo das diferencias na interpretación dun modelo cando os eventos xerados pola súa evolución (cambios no estado das etapas e das variábeis internas) son considerados na escala de tempo interna ou na externa. A interpretación mostrada considera un caso ideal no que a resposta do sistema é instantánea dende o punto de vista externo e, en consecuencia, non existe a posibilidade de que se reciba un novo evento do proceso mentres se realiza unha evolución do modelo.

Mais, como se comentou anteriormente, isto non é posíbel garantilo na práctica, polo que ao final dunha evolución pode haber eventos do proceso, recibidos durante a evolución do modelo, agardando a ser procesados. Se o intérprete se configura para considerar no tempo externo (na seguinte evolución externa) os eventos internos producidos durante a evolución do modelo, pode darse o caso de que ao final dunha evolución se teñan tanto eventos de proceso como eventos producidos pola evolución do modelo pendentes de ser procesados. O algoritmo *SIEPlayerAlgorithm* ten en conta simultaneamente ambos tipos de eventos, mentres que o algoritmo *NSIEPlayerAlgorithm* da prioridade aos eventos producidos pola evolución do modelo. O pseudocódigo seguinte mostra a diferencia entre ambos algoritmos:

```
2825. // SIEPlayerAlgorithm
2826. events = ∅
2827. while (!end)
2828.    process_events = getInputEvents()
2829.    events = events + process_events
2830.    model_events = evolveGame(events)
2831.    events = model_events
2832. end while

2833. // NSIEPlayerAlgorithm
2834. events = ∅
2835. while (!end)
2836.    process_events = getInputEvents()
2837.    events = process_events
2838.    while (!events.empty())
2839.       model_events = evolveGame(events)
2840.       events = model_events
2841.    end while
2842. end while
```

8.3.1.4. A política de evolución

A política de evolución, representada na Figura 8.3 mediante a clase *EvolutionPolicy*, é a que realiza a evolución dun modelo Grafcet dende a súa situación actual á seguinte tendo en conta os parámetros de configuración do intérprete, os valores das variábeis do proceso e os eventos detectados. Como pode verse na Figura 8.3 na versión actual da máquina virtual inclúense dúas políticas de evolución diferentes representadas mediante as clases *WoSR* e *WSR*, que se corresponden cos algoritmos de evolución sen e con busca de estabilidade (§3.3.2.2). Nótese que para permitir o reaproveitamento das políticas de evolución aplicouse o patrón de deseño "Decorator" [67], definíndose dous tipos de políticas: as simples, instancias da clase *SimpleEvolutionPolicy*, e as compostas, instancias da clase *CompoundEvolutionPolicy*. As compostas son implementadas mediante composición reaproveitando as políticas simples ou compostas xa existentes. A política *WSR* é un exemplo de política composta implementada reaproveitando a política simple *WoSR*.

8.3.1.5. A política de execución

A política de execución, representada na Figura 8.3 mediante a clase *RunningPolicy*, é a que xestiona a execución do código do modelo Grafcet: avaliación de condicións, execución de accións e control de temporizacións; así como o acceso aos servicios do núcleo da máquina virtual. A clase *DefaultRunningPolicy* proporciona a implementación por defecto desta política que é explica con detalle en (§8.6).

8.3.2. Parámetros de interpretación dos modelos

As configuracións que utilizan unha política de evolución tipo *WSR* manexan a situación do modelo en dúas escalas temporais diferentes, a correspondente ao punto de vista externo (do proceso) e a do punto de vista interno (do sistema de control). Como se explica en (§3.3.2.6), a utilización de dúas escalas de tempo permite diferentes alternativas segundo se consideren os eventos, valores de variábeis e accións internas. O intérprete permite configurar estes aspectos mediante tres parámetros que son pasados como argumentos do método *playModel* (liña 2807) cando se inicia a execución dun modelo. Os valores destes parámetros indican:

1. Se os eventos externos son considerados como eventos ou como constantes na escala interna.
2. Se os cambios nas variábeis e estados das etapas do modelo despois dunha evolución interna son tomados en consideración na seguinte evolución interna ou na seguinte evolución externa.
3. Se utilizar ou non accións internas —incluídas as ordes de forzado (§3.3.2.5)— durante as evolucións internas do modelo.

A Táboa 8-I resume as diferentes configuracións que pode adoptar o intérprete utilizando os tipos de módulos e parámetros explicados.

8.4. Funcionamento do intérprete

Neste apartado descríbense as principais colaboracións entre os módulos que forman a estructura do intérprete coas que se implementa a interpretación de modelos Grafcet.

8.4.1. Ciclo de execución do intérprete

Na Figura 8.5 móstrase a secuencia de mensaxes da execución paso a paso dun modelo Grafcet. O modelo deberá estar cargado na memoria do intérprete mediante unha chamada previa ao método *loadModel* (liña 2805). A execución iníciase coa invocación da versión con notificación asíncrona do método *playModel* (liña 2813), en resposta ao envío dunha mensaxe PLAY_MODEL dende un cliente remoto. Este método crea e inicia unha nova interpretación (xogo) do modelo, almacena os parámetros da notificación asíncrona e desbloquea o intérprete. A execución do ciclo realízase no método *OnEvent* (liña 2032), que obtén do xestor de eventos os novos eventos de entrada —método *getInputEvents*— e almacénaos no xogo —método *storeInputEvents* (liña 2859)—, delega a evolución do modelo no algoritmo de interpretación —método *evolveGame*—, e actualiza, mediante o xestor de saída, os novos valores calculados para as saídas do proceso —método *setOutputValues*—.

Ao remate da execución do ciclo envíaselle unha notificación asíncrona á máquina virtual que á súa vez envía unha mensaxe OK ao cliente remoto para indicarlle este feito. O seguinte ciclo non comeza ata que a máquina virtual desbloquee de novo ao intérprete invocando o seu método *nextModelEvolution* (liña 2821), en resposta a unha mensaxe NEXT_MODEL_EVOLUTION recibida dende o cliente remoto. A execución do modelo remata coa invocación do método *stopModel* (liña 2822) que finaliza o xogo e o elimina da memoria do intérprete. Nótese que a secuencia de mensaxes da versión continua sería semellante á explicada coa excepción da notificación asíncrona e o intercambio de mensaxes OK e NEXT_MODEL_EVOLUTION co cliente remoto.

Configuración		
Nome	Módulos	Parámetros
WOSR_FIRST	Acceso = *GatherFirstPolicy* Reacción = *DefaultReactivenessPolicy* Algoritmo = *SIEPlayerAlgorithm* Evolución = *WoSR* Execución = *DefaultRunningPolicy*	N/A
WORST_ALL	Acceso = *GatherAllPolicy* Reacción = *DefaultReactivenessPolicy* Algoritmo = *SIEPlayerAlgorithm* Evolución = *WoSR* Execución = *DefaultRunningPolicy*	N/A
WSR_SIE	Acceso = *GatherFirstPolicy* Reacción = *DefaultReactivenessPolicy* Algoritmo = *SIEPlayerAlgorithm* Evolución = *WSR* Execución = *DefaultRunningPolicy*	Eventos externos = eventos/constantes internas Valores internos = evolución interna/externa Ordes internas = Si/Non
WSR_NSIE	Acceso = *GatherFirstPolicy* Reacción = *DefaultReactivenessPolicy* Algoritmo = *NSIEPlayerAlgorithm* Evolución = *WSR* Execución = *DefaultRunningPolicy*	Eventos externos = eventos/constantes internas Valores internos = evolución interna/externa Ordes internas = Si/Non

Táboa 8-I. Configuracións do intérprete Grafcet.

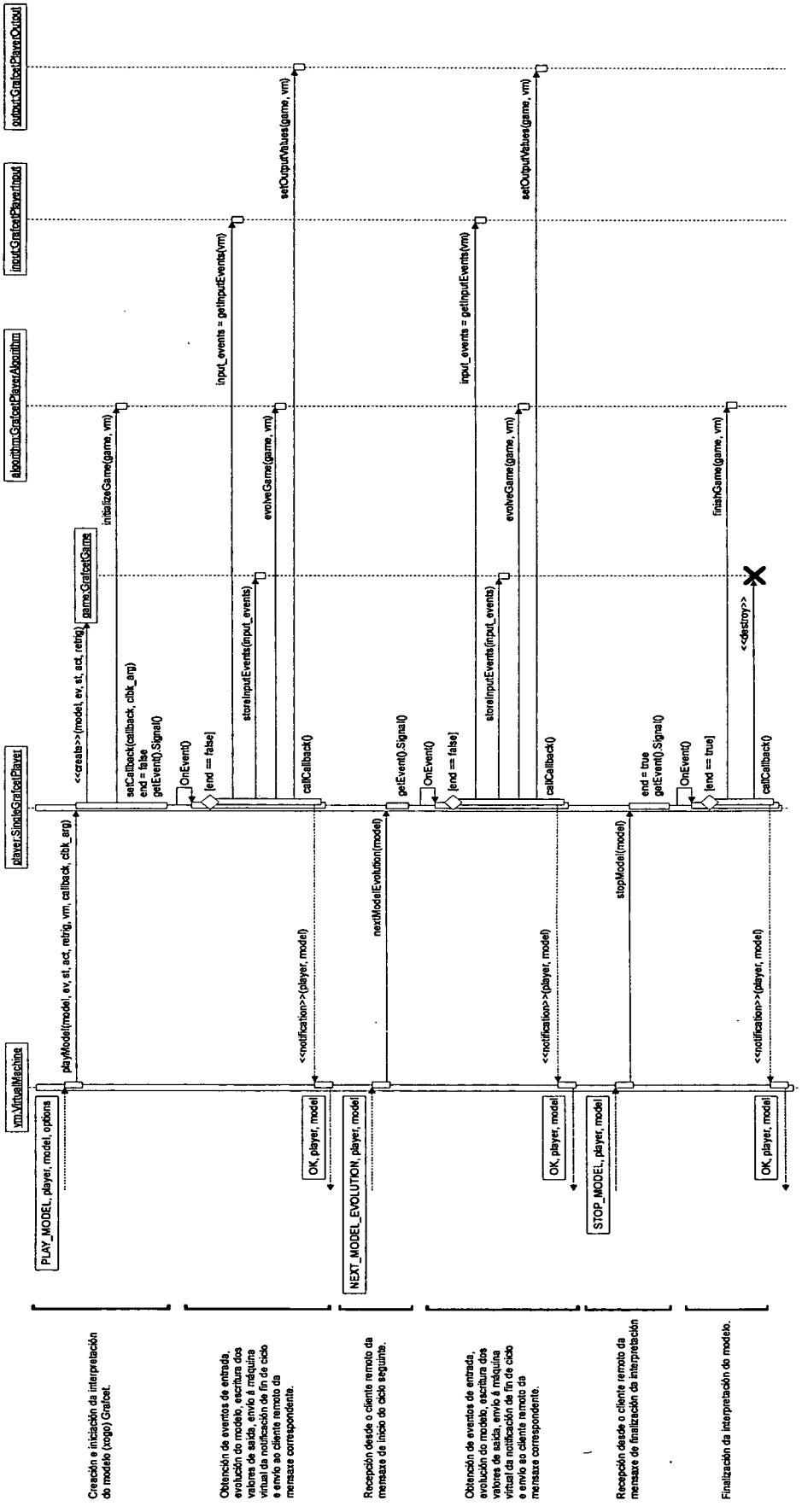


Figura 8.5. Secuencia de mensaxes do ciclo de execución do intérprete Grafcet.

8.4.2. Obtención de eventos de entrada

A Figura 8.6 mostra as mensaxes da colaboración para a obtención de novos eventos de entrada. Cando é invocado o método *getInputEvents* no xestor de eventos, este consulta a existencia de eventos almacenados na política de acceso e, se non hai ningún, obtén os eventos almacenados no canal de entrada do subsistema de aplicación. En caso de haber novos eventos estes serán almacenados na política de acceso mediante o método *onInputEvents*, que filtra os non significativos e os non válidos (§8.3.1.1). O manexo dos eventos filtrados realízase na política de reacción —métodos *onUnmanaged* e *OnUnexpected*, respectivamente—. A versión por defecto destes métodos non filtra ningún evento, considerando que todos son significativos e válidos. Por último repítese a consulta á política de acceso que agora almacena os novos eventos obtidos do canal de entrada. Nótese que nesta implementación os eventos de entrada non son procesados mentres aínda haxa eventos dos previamente almacenados na política de acceso (procesamento por antigüidade).

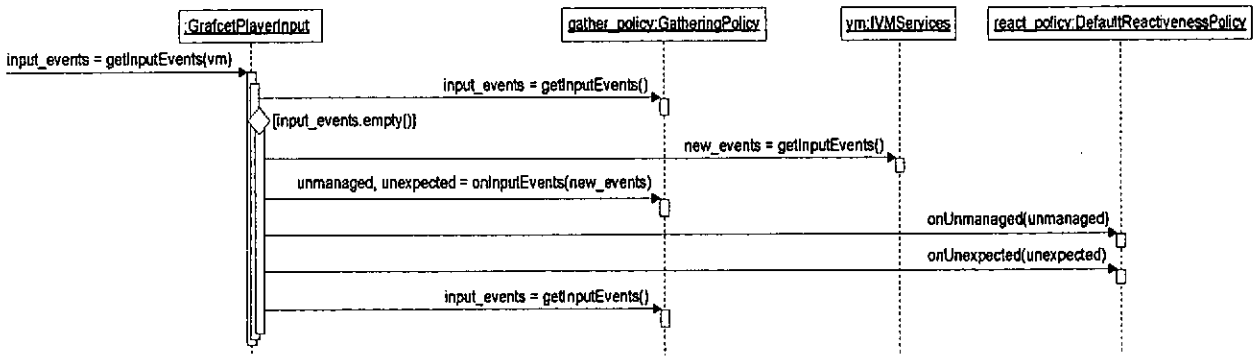


Figura 8.6. Secuencia de mensaxes da obtención de novos eventos de entrada.

8.4.3. Iniciación e evolución do modelo

A clase *GrafcetPlayerAlgorithm*, derivada da interface *IGrafcetPlayerAlgorithm* (Figura 8.3), proporciona a implementación por defecto dos métodos *initializeGame* e *finishGame*, que permiten iniciar e finalizar a interpretación de modelos Grafcet. A Figura 8.7 mostra a secuencia de mensaxes da colaboración que implementa o primeiro destes métodos. A iniciación realízase activando as etapas da situación inicial do modelo —regra 1 de evolución (§3.3.1.1)—, executando as accións internas activas na situación inicial e facendo a evolución inicial do modelo considerando os eventos xerados durante a activación da situación inicial. No que respecta ao método para finalizar a interpretación, non ten implementación na versión actual.

A Figura 8.8 mostra a secuencia de mensaxes da implementación do método *evolveGame* na clase *SIEPlayerAlgorithm*, que modela un tipo de algoritmo de evolución que considera simultaneamente os eventos producidos pola evolución do modelo e os detectados no proceso durante a evolución (§8.3.1.3). O método realiza a evolución externa do modelo actualizando no xogo Grafcet os eventos a considerar —método *updateEvents* (liña 2861)—, realizando as operacións relacionadas coa xestión dos temporizadores —método *calculateTimers*—, delegando na política de evolución a evolución do modelo —método *evolve*—, almacenando no xogo Grafcet os novos eventos xerados pola evolución —método *storeNewEvents* (liña 2862)— e delegando na política de execución a execución das accións activas —método *run*—. A implementación deste mesmo método na clase *NSIEPlayerAlgorithm* consiste nun bucle que repite a secuencia anterior ata que non haxa mais eventos producidos pola evolución do modelo.

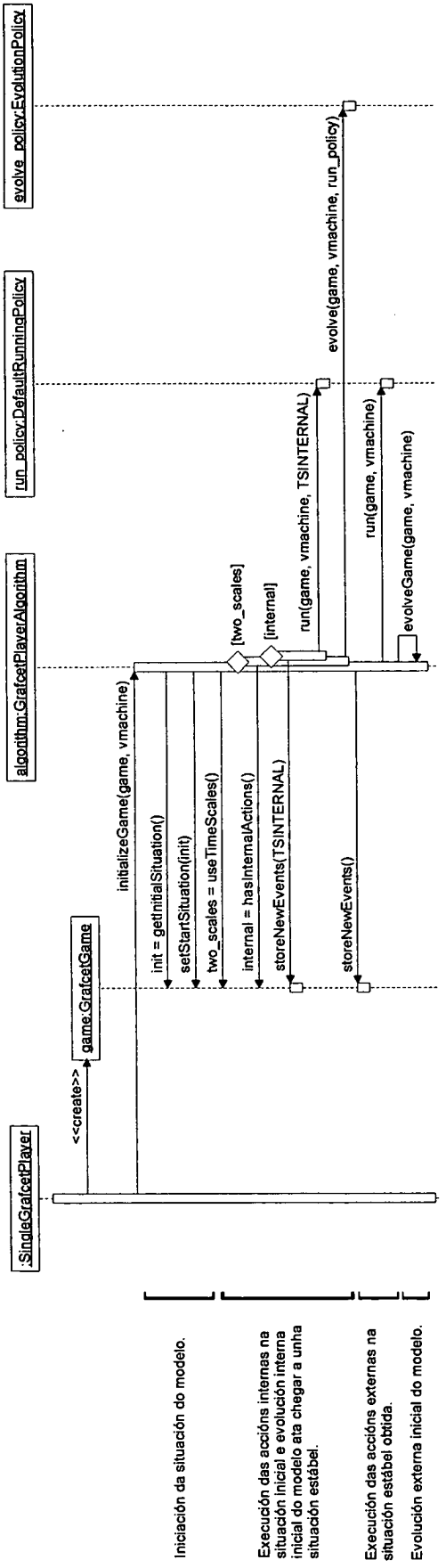


Figura 8.7. Secuencia de mensaxes da evolución inicial dun modelo Grafcet.

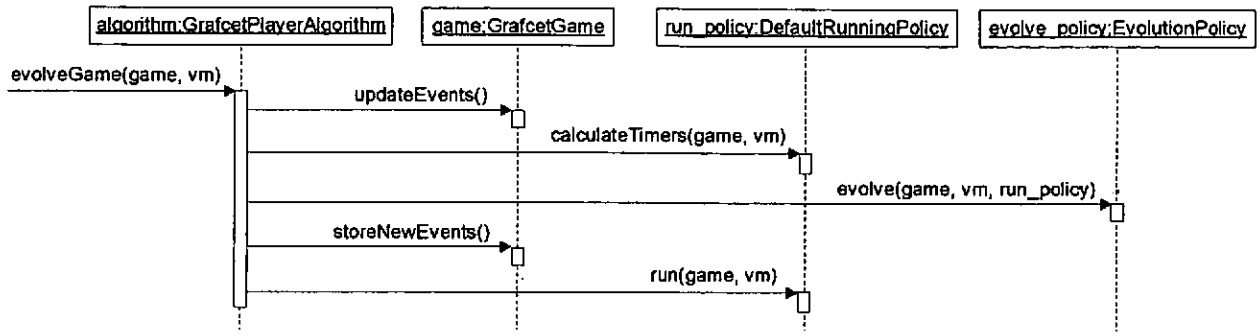


Figura 8.8. Secuencia de mensaxes da evolución externa dun modelo Grafcet.

8.4.4. Actualización das saídas do proceso

A Figura 8.9 mostra a secuencia de mensaxes mediante a que se realiza a actualización na base de datos de E/S dos valores das saídas do proceso modificados como resultado da evolución do modelo. O método *setOutputValues* na clase *GrafcetPlayerOutput* accede aos valores modificados almacenados no xogo Grafcet e, para cada un deles, envía unha mensaxe á porta de saída do subsistema de aplicación. Estas mensaxes son procesadas pola base de datos de E/S, que actualiza a imaxe do proceso cos valores recibidos. A sincronización desta imaxe co proceso (§7.3.1.3) realízase cando o intérprete invoca o método *setOutputValues* (liña 2661) dos servicios do núcleo da máquina virtual.

8.4.5. Política de evolución

A política de evolución é responsábel da evolución da situación dos modelos. Como se explica en (§8.3.1.4), na versión actual do intérprete existen dúas posibilidades: a política *WoSR*, que realiza unha evolución sen busca de estabilidade (SRS), limitándose a aplicar as regras de evolución 2 á 5 (§3.3.1) avaliando as transicións validadas e calculando a nova situación despois do franqueamento simultáneo das que cumpran a condición de transición; e a política *WSR*, que reutiliza a política *WoSR* engadíndolle a comprobación da estabilidade da nova situación, a detección de ciclos estacionarios e a execución de accións internas. A Figura 8.10 mostra a secuencia de mensaxes da colaboración que implementa o método *evolve* na política *WSR*.

8.4.6. Detección de ciclos estacionarios

Cando se utiliza o algoritmo de interpretación con busca de estabilidade (ARS) é preciso detectar a existencia de ciclos estacionarios para evitar que o modelo evolucione sen alcanzar nunca unha situación estábel (§3.3.2.2). O algoritmo descrito en [25] baséase no almacenamento das situacións internas polas que evoluciona o modelo detectando a existencia de situacións duplicadas. Sen embargo, como se indica en [76], a utilización de variábeis non booleanas, cuxos valores podan ser modificados e consultados durante as evolucións internas do modelo, pode dar lugar a que ese algoritmo detecte incorrectamente ciclos estacionarios en evolucións válidas.

Considérese o exemplo mostrado na Figura 8.11, na que as accións internas son indicadas escribindo o seu tipo entre corchetes e a interpretación utilizada considera os cambios nos valores internos das variábeis na seguinte evolución interna. Neste exemplo, despois da detección do evento $\uparrow a$, execútase a acción interna asociada á desactivación da etapa i-1 que lle asigna á variábel c o valor de k. Se k 5 a evolución do modelo será {i-1}→(i)→(i+2)→..., sen

embargo se $k<5$, o modelo evolucionará a través do ciclo (i)→(i+1) e o algoritmo detectará a existencia dun ciclo estacionario {i-1}→(i)→(i+1)→(i)→..., o cal non é correcto xa que a situación volvese estábel despois dun número de iteracións igual a $5-k$, supoñendo $k\geq 0$. Isto implica que o resultado do algoritmo de detección de ciclos estacionarios dependerá do valor de k.

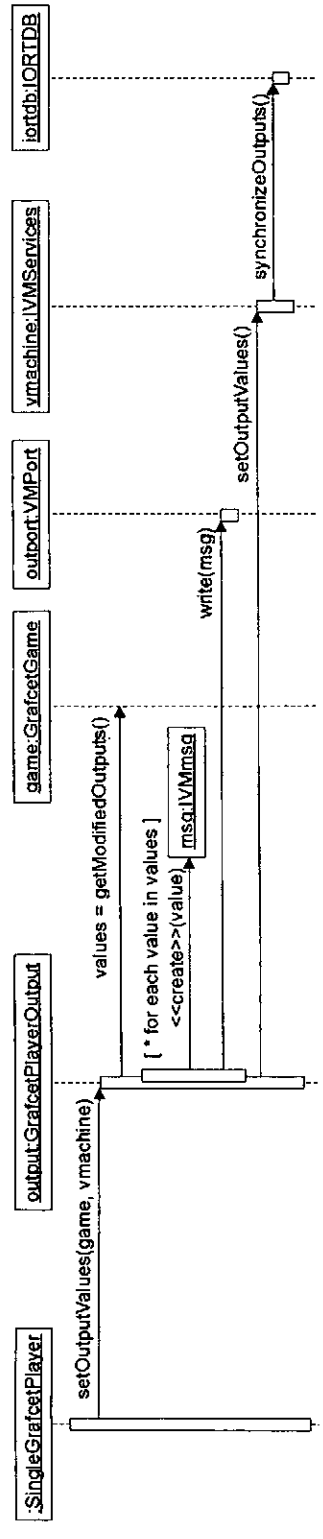


Figura 8.9. Secuencia de mensaxes da actualización das saídas do proceso.

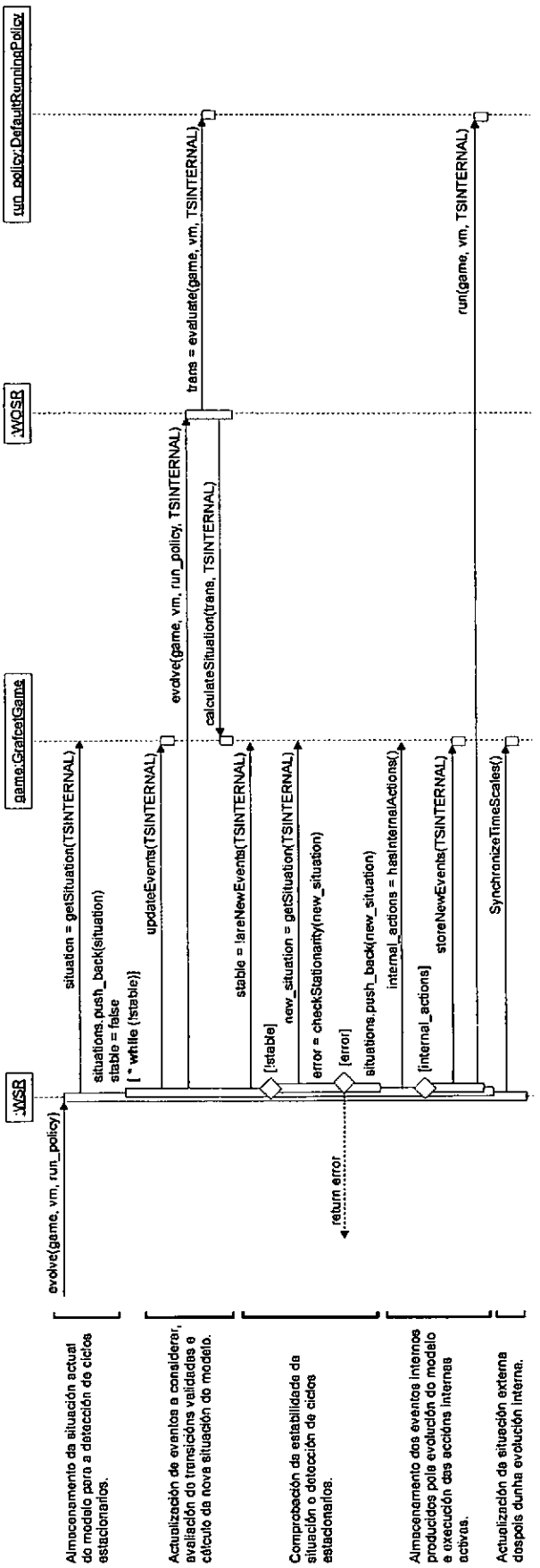


Figura 8.10. Secuencia de mensaxes da evolución interna dun modelo Grafcet.

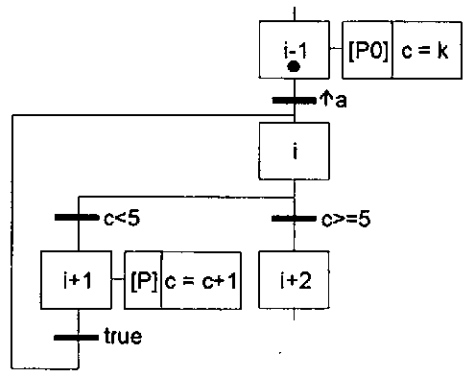


Figura 8.11. Grafcet utilizado como exemplo.

Aínda non existe un método que permita solucionar este problema, requiríndose unha análise a priori de cada modelo concreto para evitar situacións como a anterior. En [76] proponse como regra xeral considerar que non existe un ciclo estacionario se o modelo chega por segunda vez a unha mesma situación cando esta sexa estábel. Esta proposta non soluciona o problema mais é coherente co algoritmo clásico. A versión actual do intérprete utiliza esta aproximación.

8.5. Información de interpretación dun modelo: o xogo Grafcet

A información utilizada para a interpretación dun modelo Grafcet é representada mediante a clase *GrafcetGame*. O intérprete almacena o estado de execución de cada modelo utilizando instancias desta clase. A versión actual do intérprete unicamente permite realizar unha única interpretación dun único modelo, sen embargo a súa arquitectura modular (Figura 8.1) e a separación entre os algoritmos e a información de interpretación dos modelos, facilitará a implementación en futuras versións de intérpretes que xestionen múltiples interpretacións dun ou varios modelos diferentes.

A Figura 8.12 mostra unha versión simplificada das relacións de agregación da clase *GrafcetGame*. A información mantida en cada instancia da clase durante a interpretación dun modelo é a seguinte:

1. Un identificador alfanumérico único —atributo *name*—.
2. Os parámetros de interpretación do modelo (§8.3.2) —atributos *eventScale*, *stepScale* e *internalActions*—.
3. A información do modelo no formato de execución (§6.4) xerado polo compilador Grafcet —agregación *model*—.
4. Os eventos de entrada considerados na evolución actual do modelo —agregación *input_events*—.
5. A información sobre as transicións, accións, ordes de forzado e temporizadores do modelo —agregacións *receptivities*, *actions*, *forders* e *timers*, respectivamente—. As clases *ReceptivityInfo*, *ActionInfo*, *ForcingOrderInfo* e *TimerInfo* almacenan a información do estado de activación dos elementos do modelo citados e xestionan a información que permite optimizar a avaliación das condicións de transicións, asociacións, temporizadores, e a aplicación das ordes de forzado[78].

[78] Para evitar introducir aspectos innecesarios sobre a implementación destas clases non se detalla a información que conteñen, sen embargo si se explica en (§8.6) a forma en que esta información é utilizada polo intérprete.

6. A información de interpretación do modelo dependente da escala de tempo, interna ou externa, considerada —agregación *time_scales*—. Cando se utiliza un algoritmo de interpretación ARS certos aspectos da información de interpretación do modelo son diferentes en cada unhà das dúas escalas de tempo utilizadas. A información almacenada no xogo Grafcet para cada escala de tempo é representada mediante a clase *GameTimeScale*, e consiste no seguinte:
   a. A situación actual (o conxunto de etapas activas) do modelo —atributo *situation*—.
   b. conxunto de transicións validadas na situación actual —atributo *validated_trans*—.
   c. Os eventos considerados na evolución actual —agregación *events*—.
   d. Os valores das variábeis do modelo e das saídas do proceso modificadas a consecuencia da execución das accións do modelo —agregación *vars*—.
   e. O estado de activación das asociacións —agregación *associations*—.

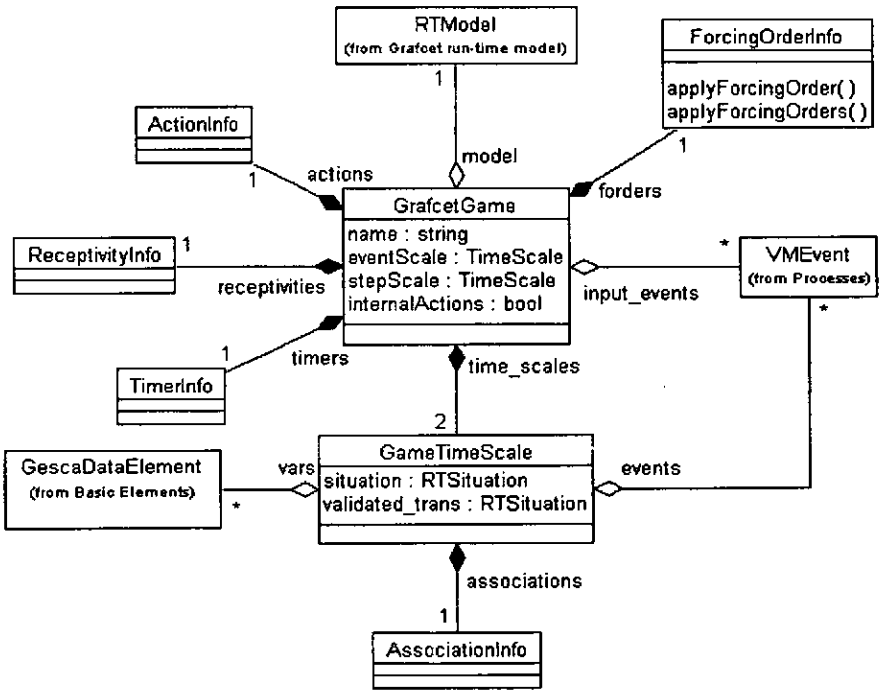


Figura 8.12. Diagrama das clases utilizadas para modelar un xogo Grafcet.

8.5.1. Implementación do xogo Grafcet

Os métodos declarados pola clase *GrafcetGame* permiten consultar a información de execución do modelo xerada polo compilador e consultar e actualizar a información do estado da interpretación do modelo. O seguinte código C++ mostra a declaración simplificada da súa interface:

```
2843. class GrafcetGame
2844. {
2845. public:
2846.   // consulta e actualización da situación do modelo
2847.   void getInitialSituation(RTSituation& s) const;
2848.   void getSituation(RTSituation& s,
2849.                     TimeScale scale = TSEXTERNAL) const;
2850.   void setStartSituation(const RTSituation& s);
2851.   void calculateSituation(const RTSituation& fired_transitions,
2852.                     TimeScale scale = TSEXTERNAL);
```

```
2853. // consulta e sincronización das escalas de tempo
2854. bool hasInternalActions() const;
2855. bool useTimeScales() const;
2856. void synchronizeTimeScales();
2857. void getModifiedOutputs(deque<GescaDataElement*>& values);
2858. // consulta e actualización de eventos
2859. void storeInputEvents(deque<VMEvent>& events);
2860. void updateTimerEvents();
2861. void updateEvents(TimeScale scale = TSEXTERNAL);
2862. void storeNewEvents(TimeScale scale = TSEXTERNAL);
2863. bool areNewEvents(TimeScale scale = TSEXTERNAL) const;
2864. // consulta e actualización de variábeis
2865. bool getStepState(unsigned long id);
2866. bool getModelVarEvent(const string& id, bool updown);
2867. bool getSystemVarEvent(const string& id, bool updown);
2868. template <class T>
2869.  bool getData(const string& id,
2870.               T& value,
2871.               TimeScale ts = TSEXTERNAL);
2872. template <class T>
2873.  bool setData(const string& id,
2874.               const T& value,
2875.               TimeScale ts = TSEXTERNAL);
2876. // acceso ás receptividades, accións e temporizadores activos
2877. void getValidatedReceptivities(deque<ReceptivityScheduleInfo>& val_recs,
2878.               TimeScale scale = TSEXTERNAL);
2879. void getActionAssociations(deque<AssociationScheduleInfo>& act_assocs,
2880.               deque<AssociationScheduleInfo>& deact_assocs,
2881.               TimeScale scale = TSEXTERNAL);
2882. void getActions(deque<ActionScheduleInfo>& action_seq);
2883. void getActiveTimers(deque<TimerScheduleInfo>& t);
2884. // acceso ás funcións co código C++ de condicións e accións
2885. pfnCond getCondition(const string& name) const;
2886. pfnCode getAction(const string& name) const;
2887.};
```

A continuación detállanse algúns aspectos relacionados coa utilización e implementación dos métodos declarados na interface pública da clase *GrafcetGame*.

8.5.1.1. Consulta e actualización da situación do modelo

Os métodos *getInitialSituation* (liña 2847) e *getSituation* (liña 2848) permiten consultar a situación inicial do modelo (o conxunto de etapas iniciais) e a situación actual almacenada na escala de tempo indicada. O método *setStartSituation* (liña 2850) establece a situación de partida das evolucións do modelo, que pode ser a inicial ou outra situación calquera. Este método é invocado durante a iniciación da interpretación do modelo (§8.4.3), e o pseudocódigo seguinte mostra as operacións realizadas na súa implementación:

```
2888.void GrafcetGame::setStartSituation(const RTSituation& s)
2889.{
2890.  scale = useTimeScales()
2891.  // aplicar ordes de forzado
2892.  if (scale && internalActions)
2893.    initial_situation = forders.applyForcingOrders(s, Ø, Ø, model->static_model)
2894.  end if
2895.  // iniciar na escala de tempo correspondente a
2896.  // información da situación actual, transicións validadas,
2897.  // e asociacións activas.
2898.  time_scales[scale].setSituation(initial_situation)
2899.  time_scales[scale].calculateValidatedTransitions(model->static_model)
2900.  time_scales[scale].updateAssociationInfo()
2901.}
```

O método comeza determinando cal é a escala de tempo a modificar, en caso de ser a escala interna e estar activadas as accións internas aplícanse as ordes de forzado activas na situación indicada, que poden modificar a situación de partida. Coa nova situación calculada actualízase a información almacenada na escala de tempo afectada, utilizando os métodos da clase *GameTimeScale* que se explican en (§8.5.2).

O método *calculateSituation* (liña 2851) é invocado dende a política de evolución (§8.4.5) para calcular a nova situación do modelo, recibindo como parámetros o conxunto de transicións a franquear na situación actual e a escala de tempo na que se realiza a evolución. O pseudocódigo seguinte mostra as operacións realizadas na súa implementación:

```
2902. void
2903.  GrafcetGame::calculateSituation(const RTSituation& fired_transitions,
2904.                                   TimeScale scale)
2905. {
2906.    current_situation = time_scales[scale].situation
2907.    // calcular etapas activadas e desactivadas
2908.    enabled = ∅
2909.    disabled = ∅
2910.    for each transition (t) in fired_transitions
2911.       enabled = enabled ∪ t.after
2912.       disabled = disabled ∪ t.before
2913.    end for
2914.    // aplicar ordes de forzado
2915.    if (scale && internalActions)
2916.       enabled, disabled = forders.applyForcingOrders(current_situation,
2917.                                                      enabled,
2918.                                                      disabled,
2919.                                                      model->static_model)
2920.    end if
2921.    // calcular nova situación
2922.    // as etapas que son activadas e desactivadas simultaneamente
2923.    // mantéñense activas (regra 5 de evolución (§3.3.1.5))
2924.    new_situation = (current_situation - disabled) ∪ enabled
2925.    // actualizar na escala de tempo correspondente a
2926.    // información da situación actual, transicións validadas,
2927.    // e asociacións activas.
2928.    time_scales[scale].setSituation(new_situation)
2929.    time_scales[scale].calculateValidatedTransitions(model->static_model,
2930.                                                     fired_transitions)
2931.    time_scales[scale].updateAssociationInfo()
2932. }
```

O método comeza determinando os conxuntos de etapas a activar e desactivar tendo en conta a situación actual do modelo e o conxunto de transicións a franquear. Se as accións internas están activadas e a evolución se realiza na escala interna, aplícanse as ordes de forzado que poden modificar os conxuntos de etapas a activar e desactivar. Unha vez aplicadas as ordes de forzado, calcúlase a nova situación e actualízase a información almacenada na escala de tempo afectada.

Nos dous métodos anteriores as ordes de forzado do modelo aplícanse mediante o método *applyForcingOrders* da clase *ForcingOrderInfo* respectando os postulados temporais explicados en (§3.3.2.5). Este método utiliza a información sobre os niveis da xerarquía de forzado xerada polo compilador Grafcet para determinar o conxunto de etapas a activar e desactivar nunha situación dada. O pseudocódigo da versión simplificada do algoritmo utilizado é o seguinte:

```
2933. void
2934.    ForcingOrderInfo::applyForcingOrders(const RTSituation& current,
2935.                                         RTSituation& disabled,
2936.                                         RTSituation& enabled,
2937.                                         const RTStaticModel& model)
2938. {
2939.    forced_partial_grafcets = ∅
2940.    // aplicar ordes respectando niveis da xerarquía de forzado
2941.    for each level (l) in model.FO_levels
2942.       // calcular situación previa á aplicación das ordes do nivel
2943.       situation_before = (current - disabled) ∪ enabled
2944.       // aplicar ordes de cada grafcet parcial "forzador" no nivel
2945.       for each partial grafcet (pg) in l
2946.          // obter etapas activas do grafcet parcial con ordes asociadas
2947.          forder_steps = pg.fosteps ∩ situation_before
2948.          for each step (s) in forder_steps
2949.             // obter ordes de forzado asociadas á etapa
2950.             active_forders = model.forcing_orders[s]
2951.             for each forcing order (fo) in active_forders
2952.                // comprobar se outra orde forza o mesmo grafcet parcial
2953.                if (fo.fpg in forced_partial_grafcets)
2954.                   indicar error (forzado múltiple)
2955.                end if
2956.                forced_partial_grafcets.insert(fo.fpg)
2957.                // aplicar orde de forzado
2958.                applyForcingOrder(fo, current, disabled, enabled, model)
2959.             end for
2960.          end for
2961.       end for
2962.    end for
2963. }
```

Como pode verse as ordes de forzado son aplicadas respectando a orde dos niveis da xerarquía de forzado. En cada nivel obtéñense e aplícanse as ordes de forzado asociadas ás etapas activas dos grafcets parciais do nivel. O algoritmo comproba que non se producen situacións de forzado múltiple (§3.3.2.5) almacenando no conxunto *forced_partial_grafcets* os identificadores dos grafcets parciais forzados polas ordes que van sendo aplicadas e comprobando que non existan duplicados antes de aplicar unha nova orde. A aplicación de cada orde realízase mediante o método privado *applyForcingOrder* da clase *ForcingOrderInfo*, que modifica os conxuntos de etapas a activar e desactivar como resultado da evolución do modelo. O pseudocódigo da versión simplificada deste método é o seguinte:

```
2964. void
2965.    ForcingOrderInfo::applyForcingOrder(const RTFOInfo& fo
2966.                                        const RTSituation& current,
2967.                                        RTSituation& disabled,
2968.                                        RTSituation& enabled,
2969.                                        const RTStaticModel& model)
2970. {
2971.    // obter información do grafcet parcial forzado
2972.    forced_grafcet = model.partial_grafcets[fo.fpg]
2973.    // calcular etapas forzadas no grafcet parcial segundo o tipo de orde
2974.    if (fo.type == FORCE_EMPTY)
2975.       forced_steps = ∅
2976.    else if (fo.type == FORCE_INITIAL)
2977.       // etapas do grafcet parcial contidas na situación inicial do modelo
2978.       forced_steps = model.initial_situation ∩ forced_grafcet.steps
2979.    else if (fo.type == FREEZE)
2980.       // etapas do grafcet parcial contidas na situación actual do modelo
2981.       forced_steps = current ∩ forced_grafcet.steps
2982.    else if (fo.type == FORCE_SITUATION)
2983.       forced_steps = fo.fsituation
2984.    end if
```

```
2985.   // calcular etapas non forzadas no grafcet parcial
2986.   unforced_steps = forced_grafcet.steps - forced_steps
2987.   // modificar as etapas a activar e desactivar
2988.   disabled = (disabled - forced_steps) ∪ (unforced_steps ∩ current)
2989.   enabled = (enabled - unforced_steps) ∪ forced_steps
2990. }
```

As modificacións realizadas por este método no conxunto de etapas a desactivar consisten en eliminar del as etapas que pasan a estar forzadas e engadirlle as que estando activas na situación actual deixan de estalo ao non pertencer ao conxunto de etapas forzadas. Do mesmo xeito, no conxunto de etapas a activar engádense as etapas forzadas e elimínanse as que non pertencen ao conxunto de etapas forzadas.

8.5.1.2. Consulta e sincronización das escalas de tempo

Os métodos *hasInternalActions* (liña 2854) e *useTimeScales* (liña 2855) permiten consultar respectivamente se as accións internas, incluídas as ordes de forzado, están habilitadas na interpretación do modelo e se o algoritmo de interpretación utiliza unha ou dúas escalas de tempo. O método *synchronizeTimeScales* (liña 2856) actualiza os datos almacenados na escala externa facéndoos iguais aos da escala interna. Este método é invocado dende as políticas de evolución ARS cando se chega a unha situación estábel despois dunha evolución interna do modelo. O método *getModifiedOutputs* (liña 2857) devolve os valores das saídas modificadas durante a evolución do modelo. Este método é invocado polo xestor de saída para a actualización da base de datos de E/S ao final de cada ciclo de execución do intérprete (§8.4.4).

8.5.1.3. Consulta e actualización de eventos

Os métodos desta categoría permiten xestionar os diferentes tipos de eventos externos e internos que son considerados durante as evolucións do modelo. O método *storeInputEvents* (liña 2859) é invocado ao comezo de cada ciclo de execución do intérprete (Figura 8.5) para almacenar no xogo os eventos detectados no proceso e recibidos a través do canal de entrada do subsistema de aplicación da máquina virtual. O método *updateTimerEvents* (liña 2860) é invocado para almacenar no xogo os eventos producidos a consecuencia da avaliación dos temporizadores do modelo (Figura 8.18). O método *updateEvents* (liña 2861) é invocado antes de cada evolución do modelo, xa sexa interna (Figura 8.10) ou externa (Figura 8.8), para calcular os eventos a considerar, que dependerán dos eventos de entrada, de temporización, dos producidos a consecuencia da evolución anterior do modelo e dos parámetros utilizados na interpretación (§8.3.2). O método *storeNewEvents* (liña 2862) é invocado despois de cada evolución do modelo, xa sexa interna (Figura 8.10) ou externa (Figura 8.8), e antes de executar as accións para considerar os eventos producidos a consecuencia da evolución do modelo na avaliación das condicións das asociacións activas. Por último, o método *areNewEvents* (liña 2863) sirve para consultar se se produciron novos eventos a consecuencia da evolución do modelo. Este método é utilizado nas políticas de evolución ARS para detectar as situacións estábeis (Figura 8.10).

8.5.1.4. Consulta e actualización de variábeis

Os métodos desta categoría proporcionan acceso ao estado de activación das etapas —método *getStepState* (liña 2865)—, aos eventos considerados durante a evolución —métodos *getModelVarEvent* (liña 2866) e *getSystemVarEvent* (liña 2867)— e aos valores das variábeis do modelo —métodos *getData* (liña 2869) e *setData* (liña 2873)—. Estes métodos son invocados pola política de execución (§8.6) durante a execución das funcións que conteñen o

código C++ das condicións e accións do modelo. Nótese que a escala de tempo utilizada polos métodos *getStepState*, *getModelVarEvent* e *getSystemVarEvent* non se pasa como argumento senón que é calculada internamente en función dos parámetros de interpretación do modelo (§8.3.2).

8.5.1.5. Acceso ás receptividades, accións e temporizadores activos

Os métodos desta categoría son invocados pola política de execución para obter a información sobre as transicións validadas —método *getValidatedReceptivities* (liña 2877)—, e as asociacións —método *getActionAssociations* (liña 2879)—, accións —método *getActions* (liña 2882)— e temporizadores activos —método *getActiveTimers* (liña 2883)— para a avaliación de receptividades (§8.6.1), temporizadores (§8.6.2), e a execución de accións (§8.6.3). A información devolta está optimizada para reducir o número de operacións a realizar dacordo á técnica explicada en (§8.6.1).

8.5.1.6. Acceso ás funcións co código C++ de condicións e accións

Os métodos desta categoría tamén son invocados pola política de execución (§8.6) para acceder ás funcións que conteñen o código C++ das condicións —método *getCondition* (liña 2885)— e accións —método *getAction* (liña 2886)— do modelo.

8.5.2. Información para cada escala de tempo

Como se comentou anteriormente, a clase *GameTimeScale* almacena a información que depende da escala de tempo considerada cando se utiliza un algoritmo de interpretación ARS. Os métodos desta clase permiten consultar e actualizar a información da escala de tempo, a declaración simplificada da súa interface é a seguinte:

```
2991.class GameTimeScale
2992.{
2993.public:
2994.   // consulta e actualización de variábeis
2995.   bool getStepState(unsigned long id);
2996.   bool getEvent(const string& id, bool updown);
2997.   template <class T>
2998.     bool getData(const string& id, T& value);
2999.   template <class T>
3000.     bool setData(const string& id, const T& value);
3001.   // actualización da información da escala de tempo
3002.   void setSituation(const RTSituation& new_situation);
3003.   void calculateValidatedTransitions(const RTStaticModel& model,
3004.                        const RTSituation& fired_transitions = RTSituation());
3005.   void updateAssociationInfo();
3006.};
```

Os métodos de consulta e actualización de variábeis proporcionan a mesma funcionalidade que os da clase *GrafcetGame*, permitindo o acceso na escala de tempo ao estado de activación das etapas —método *getStepState* (liña 2995)—, aos eventos considerados durante a evolución —método *getEvent* (liña 2996)— e aos valores das variábeis do modelo —métodos *getData* (liña 2998) e *setData* (liña 3000)—.

Os demais métodos permiten actualizar a información almacenada pola escala de tempo cando se establece unha nova situación do modelo. O método *setSituation* (liña 3002) almacena a nova situación e calcula e almacena os novos eventos xerados pola evolución. O pseudocódigo simplificado da súa implementación é o seguinte:

```
3007. void GameTimeScale::setSituation(const RTSituation& new_situation)
3008. {
3009.    // calcular etapas activadas e desactivadas
3010.    disabled = situation - new_situation
3011.    enabled = new_situation - situation
3012.    // almacenar eventos xerados pola activación e desactivación de etapas
3013.    for each step (s) in disabled
3014.      events.insert(VMEvent(s, false))
3015.    end for
3016.    for each step (s) in enabled
3017.      events.insert(VMEvent(s, true))
3018.    end for
3019.    // almacenar a nova situación
3020.    situation = new_situation
3021. }
```

O método *calculateValidatedTransitions* (liña 3003) calcula as transicións validadas na nova situación da escala de tempo (a establecida polo método anterior) tendo en conta o conxunto de transicións franqueábeis calculado pola política de evolución (§8.4.5). Nótese que a situación almacenada na escala de tempo é a obtida despois de aplicar as ordes de forzado (liñas 2915-2920), polo que o conxunto de transicións realmente franqueadas non ten necesariamente que coincidir coas transicións franqueábeis obtidas pola política de evolución como resultado da evolución estructural do modelo. Este aspecto é considerado na implementación deste método, cuxo pseudocódigo simplificado é o seguinte:

```
3022. void
3023.  GameTimeScale::calculateValidatedTransitions(const RTStaticModel& model,
3024.                                                const RTSituation& fired_transitions)
3025. {
3026.    // calcular etapas activadas e desactivadas
3027.    disabled = situation - new_situation
3028.    enabled = new_situation - situation
3029.    // calcular transicións que deixan de estar validadas
3030.    // (as sucesoras das etapas desactivadas)
3031.    invalidated = ∅
3032.    for each step (s) in disabled
3033.      invalidated = invalidated ∪ s.after
3034.    end for
3035.    // calcular transicións que poden pasar a estar validadas
3036.    // (as sucesoras das etapas activadas)
3037.    candidate = ∅
3038.    for each step (s) in enabled
3039.      candidate = candidate ∪ s.after
3040.    end for
3041.    // comprobar que transicións están realmente validadas
3042.    // (unha transición está validada cando todas as etapas
3043.    //  que a preceden están activas na situación actual)
3044.    new_validated = ∅
3045.    for each transition (t) in candidate
3046.      steps_before = t.before
3047.      if ((steps_before ∩ situation) == steps_before)
3048.        new_validated.insert(t)
3049.      end if
3050.    end for
3051.    // engadir ás transicións validadas as transicións franqueábeis que non
3052.    // son finalmente franqueadas debido á aplicación das ordes de forzado
3053.    new_validated = new_validated ∪ (fired_transitions - invalidated)
3054.    // calcular e almacenar as transicións validadas na situación actual
3055.    validated_trans = (validated_trans - invalidated) ∪ new_validated
3056. }
```

Por último o método *updateAssociationInfo* (liña 3005) actualiza o estado de activación das asociacións do modelo tendo en conta a nova situación establecida na escala de tempo.

8.6. A política de execución

A política de execución é a responsábel de xestionar a execución do código do modelo Grafcet: avaliación de condicións, execución de accións e control de temporizacións; así como de proporcionar o acceso aos valores das variábeis do modelo e do proceso dende as funcións C++ xeradas polo compilador (§6.5.1.9). A clase *RunningPolicy* (Figura 8.3) implementa polo tanto dúas interfaces diferentes, a utilizada polo algoritmo de interpretación e a utilizada no código C++ de accións e condicións, que é definida pola clase abstracta *IVMachineAccess* (§6.3.2). O código seguinte mostra unha versión simplificada da súa declaración:

```
3057. class RunningPolicy : public IModule, public IVMachineAccess
3058. {
3059.    // métodos utilizados polo algoritmo de interpretación
3060.    virtual void run(GrafcetGame* game,
3061.                     IVMServices* vmachine,
3062.                     TimeScale time_scale = TSEXTERNAL) = 0;
3063.    virtual void evaluate(GrafcetGame* game,
3064.                          IVMServices* vmachine,
3065.                          RTSituation& fired_transitions,
3066.                          TimeScale time_scale = TSEXTERNAL) = 0;
3067.    virtual void calculate_timers(GrafcetGame* game,
3068.                                  IVMServices* vmachine) = 0;
3069.    // métodos utilizados no código das funcións xeradas polo compilador Grafcet
3070.    bool getModelVarEvent(const string& id, bool updown);
3071.    bool getSystemVarEvent(const string& id, bool updown);
3072.    bool getStepState(unsigned long id);
3073.    bool getTimerState(const string& id);
3074.    template <class T>
3075.     bool getModelVar(const string& id, T& value);
3076.    template <class T>
3077.     bool setModelVar(const string& id, const T& value);
3078.    template <class T>
3079.     bool getSystemVar(const string& id, T& value);
3080.    template <class T>
3081.     bool setSystemVar(const string& id, const T& value);
3082.    // métodos para a xestión das temporizacións
3083.    bool setTimerState(const string& id, bool state);
3084.    unsigned setTimer(long time, pfn callback, IAsyncClbkArg* parg = NULL);
3085.    // métodos de acceso ao código de condicións e accións no xogo Grafcet
3086.    pfnCond getCondition(const string& name) const;
3087.    pfnCode getAction(const string& name) const;
3088. };
```

A clase *RunningPolicy* proporciona a implementación por defecto dos métodos declarados na interface *IVMachineAccess* que acceden aos valores das variábeis do proceso almacenadas na base de datos de E/S —métodos *getSystemVar* (liña 3079) e *setSystemVar* (liña 3081)—, aos valores das variábeis declaradas no modelo —métodos *getModelVar* (liña 3075) e *setModelVar* (liña 3077)—, aos eventos externos e internos —métodos *getModelVarEvent* (liña 3070) e *getSystemVarEvent* (liña 3071)— e ao estado de temporizadores e etapas do modelo —métodos *getTimerState* (liña 3073) e *getStepState* (liña 3072)—. Tamén declara os métodos utilizados para o cálculo dos temporizadores —método *calculate_timers* (liña 3067)—, a avaliación das receptividades —método *evaluate* (liña 3063)— e a execución das accións —método *run* (liña 3060)—, que son implementados na clase derivada *DefaultRunningPolicy* (Figura 8.3). Ademais inclúe algúns métodos auxiliares para facilitar a xestión das temporizacións (liñas

3083 e 3084) e o acceso ao código das condicións e accións do modelo Grafcet que se estea executando (liñas 3086 e 3087).

No resto deste apartado explícanse de forma simplificada os detalles da implementación destes métodos e as técnicas empregadas para reducir as operacións a realizar en cada ciclo de evolución do modelo.

8.6.1. Avaliación e franqueamento de transicións

Para cada transición do modelo Grafcet, o xogo Grafcet almacena unha instancia da clase *ReceptivityScheduleInfo* —estas instancias son manexadas na implementación da clase *ReceptivityInfo* (Figura 8.12)—. O código da declaración simplificada desta clase é o seguinte:

```
3089. class ReceptivityScheduleInfo
3090. {
3091. private:
3092.    RTReceptivityInfo* rec;
3093. public:
3094.    bool evaluate(RunningPolicy* env) const;
3095. };
```

Cada instancia desta clase mantén un apuntador á información da transición xerada polo compilador (§6.4) e proporciona un método para avaliala. A implementación deste método é a seguinte:

```
3096. bool ReceptivityScheduleInfo::evaluate(RunningPolicy* env) const
3097. {
3098.    if (rec->receptivity.empty())
3099.       return true;
3100.    end if
3101.    return (env->getCondition(rec->receptivity))(*env);
3102. }
```

O método devolve *true* se a transición non ten receptividade asociada. En caso contrario devolve o resultado de executar a función do modelo que contén o código da receptividade —accedida mediante o método *getCondition* (liña 3086) da política de execución—, pasándolle como parámetro a política de execución, que é a que proporciona o acceso aos valores de eventos, temporizadores e variábeis.

O seguinte pseudocódigo mostra unha versión simplificada da implementación do método que calcula, para unha situación dada, as receptividades franqueábeis e as devolve no argumento *fired_transitions*. Nótese que é utilizado o mesmo método tanto nas evolucións internas como nas externas do modelo, os aspectos específicos de cada escala son mantidos no xogo Grafcet.

```
3103. void DefaultRunningPolicy::evaluate(GrafcetGame* game,
3104.                                     IVMServices* vmachine,
3105.                                     RTSituation& fired_transitions,
3106.                                     TimeScale time_scale)
3107. {
3108.    validated_receptivities = game->getValidatedReceptivities(time_scale)
3109.    for each receptivity (r) in validated_receptivities
3110.       if r.evaluate(this)
3111.          fired_transitions.insert(r)
3112.       end if
3113.    end for
3114. }
```

Para reducir o número de receptividades a avaliar, o xogo Grafcet mantén a información actualizada das transicións validadas e das variábeis booleanas utilizadas en cada receptividade

(esta información é obtida polo compilador Grafcet). Utilizando esta información, para unha situación do modelo dada, unicamente se avalían as receptividades das transicións que pasaron a estar validadas como resultado da última evolución do modelo e, das que xa estaban validadas, unicamente as que conteñan variábeis booleanas para as que haxa algún evento de entrada significativo (§8.3.1.2). Como exemplo considérese o grafcet da Figura 8.13, no que as variábeis *a* e *b* son booleanas e *c* é real.

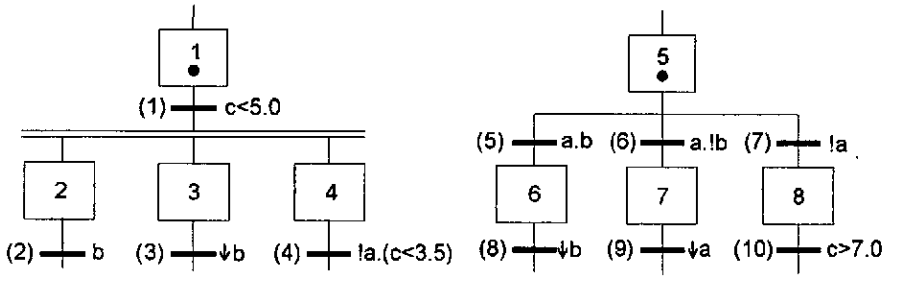


Figura 8.13. Grafcet utilizado como exemplo.

A información sobre as receptividades obtida polo compilador e almacenada no xogo Grafcet para este modelo sería a seguinte:

```
// transicións do modelo
T = {(1), (2), (3), (4), (5), (6), (7), (8), (9), (10)}
// transicións con receptividades que só conteñen variábeis booleanas
// (indexadas polo nome da variábel)
Tb = { {"a", (5), (6), (7), (9)},
       {"b", (2), (3), (5), (6), (8)} }
// transicións con receptividades que conteñen variábeis non booleanas
Tr = {(1), (4), (10)}
```

Supóñase que os valores das variábeis son: *a*=1, *b*=0 e *c*=12.5, e que o modelo está no medio dunha evolución na que ven de activarse a etapa 5. A información almacenada no xogo sobre o estado da evolución será a seguinte:

```
// situación actual
S = {1, 5}
// transicións validadas
Tv = {(1), (5), (6), (7)}
// novas transicións validadas a consecuencia da evolución anterior
Tvn = {(5), (6), (7)}
```

Se se utiliza unha política de evolución ARS, na que os eventos producidos pola evolución do modelo son considerados na seguinte evolución interna e teñen prioridade sobre os eventos externos, a avaliación e franqueamento das transicións do modelo producirá os seguintes resultados:

```
// CICLO 1
a=1, b=0, c=12.5
// eventos
E = {↑X5}
// transicións a avaliar
Ta = Tvn(-1) ∪ (Tv(-1) ∩ (Tr ∪ Tb["X5"])) = {(1), (5), (6), (7)}
// transicións franqueábeis
Tf = {∅}
// nova situación (non hai evolución)
S = {1, 5}
// transicións validadas
Tv = {(1), (5), (6), (7)}
// transicións que pasan a estar validadas a consecuencia da evolución
Tvn = {∅}
```

Do conxunto de transicións que xa estaban validadas antes da última evolución, non se avaliarán aquelas con receptividades que conteñan unicamente variábeis booleanas para as que non haxa ningún evento. A razón é que o resultado da avaliación destas transicións non pode variar ata que cambie o valor dalgunha das variábeis booleanas da súa receptividade, e os cambios nestes valores son recibidos polo intérprete en forma de eventos. No caso concreto do ciclo 1, o conxunto de transicións a avaliar coincide co de transicións validadas. Como resultado da avaliación ningunha transición pode ser franqueada, polo que o modelo queda na mesma situación na que estaba ao comezo o ciclo.

Supóñase agora que antes do seguinte ciclo de execución do intérprete o valor da variábel *c* cambia a 4.5, nese caso o resultado da avaliación e franqueamento das transicións sería o seguinte:

```
// CICLO 2
a=1, b=0, c=4.5
// eventos
E = {Ø}
// transicións a avaliar
Ta = Tvn(-1) ∪ (Tv(-1) ∩ Tr) = {(1)}
// transicións franqueábeis
Tf = {(1)}
// nova situación
S = {2, 3, 4, 5}
// transicións validadas
Tv = {(2), (3), (4), (5), (6), (7)}
// transicións que pasan a estar validadas a consecuencia da evolución
Tvn = {(2), (3), (4)}
```

Neste ciclo non hai eventos, pois *c* non é unha variábel booleana, e das transicións validadas unicamente é preciso avaliar a que contén variábeis non booleanas na súa receptividade, pois o resultado da avaliación das outras vai seguir sendo igual ao da última vez que se avaliaron, é dicir *false*. Como resultado da avaliación franquease a transición 1 e actívanse as etapas 2, 3 e 4. A evolución continuaría do xeito seguinte:

```
// CICLO 3
a=1, b=0, c=4.5
// eventos
E = {↓X1, ↑X2, ↑X3, ↑X4}
// transicións a avaliar
Ta = Tvn(-1)∪(Tv(-1)∩(Tr∪Tb["X1"]∪Tb["X2"]∪Tb["X3"]∪Tb["X4"])) = {(2),(3),(4)}
// transicións franqueábeis
Tf = {Ø}
// nova situación (non hai evolución)
S = {2, 3, 4, 5}
// transicións validadas
Tv = {(2), (3), (4), (5), (6), (7)}
// transicións que pasan a estar validadas a consecuencia da evolución
Tvn = {Ø}
```

Neste terceiro ciclo avalíanse unicamente as transicións que veñen de ser validadas pola evolución anterior, xa que as demais conteñen nas súas receptividades unicamente variábeis booleanas non afectadas polos eventos de entrada. Por último, supóñase que antes do seguinte ciclo cambia o valor da variábel *a*. Como esta variábel é booleana o intérprete recibirá un evento a través do canal de entrada do subsistema de aplicación e a evolución do modelo sería a seguinte:

```
3171.  // CICLO 4
3172.  a=0, b=0, c=4.5
3173.  // eventos
3174.  E = {↓a}
3175.  // transicións a avaliar
3176.  Ta = Tvn(-1) ∪ (Tv(-1) ∩ (Tr ∪ Tb["a"])) = {(4), (5), (6), (7)}
3177.  // transicións franqueábeis
3178.  Tf = {(7)}
3179.  // nova situación
3180.  S = {2, 3, 4, 8}
3181.  // transicións validadas
3182.  Tv = {(2), (3), (4), (10)}
3183.  // transicións que pasan a estar validadas a consecuencia da evolución
3184.  Tvn = {(10)}
```

Neste caso, das transicións validadas serán avaliadas as de receptividades que conteñan unicamente variábeis booleanas afectadas polo cambio de *a*, máis as que conteñen variábeis non booleanas.

Existen dúas consideracións adicionais que é preciso ter en conta na técnica descrita anteriormente para a reducción do número de transicións a avaliar en cada ciclo, de xeito que poda ser aplicada en todas as situacións:

1. As transicións fonte (§3.2.2.1) do modelo. Este tipo de transicións están sempre validadas, polo que na iniciación do xogo Grafcet teñen que inserirse no conxunto de transicións validadas (T_v) no que permanecerán ata que finalice a interpretación do modelo. Polo demais manexaranse do mesmo xeito que calquera outra transición, inseríndose no conxunto T_b (liña 3119) se a súa receptividade contén unicamente variábeis booleanas ou no conxunto T_r (liña 3122) se contén algunha variábel non booleana.
2. As transicións validadas como consecuencia das ordes de forzado que deixan de estar activas. Considérese o modelo mostrado na Figura 8.14, no que a transición 10 está validada mais non poderá ser franqueada, independentemente do valor da variábel *b*, mentres a etapa 10 siga estando forzada pola orde de forzado da etapa 1. Cando se recibe o evento $\uparrow a$, a situación evoluciona a {2, 10} e valídase a transición 2. Ademais, como a etapa 10 deixa de estar forzada, debe considerarse a transición 10 como se tamén acabara de validarse para que sexa avaliada e franqueada se é o caso. Nótese que unicamente é preciso considerar este caso cando se utilice unha política de evolución ARS coas ordes internas habilitadas. Para manexalo, o xogo Grafcet mantén información sobre as etapas que están forzadas en cada situación e calcula, para cada evolución na que hai cambios nas etapas forzadas, o conxunto de transicións que suceden ás etapas que deixan de estar forzadas. As transicións dese conxunto que estean validadas serán engadidas ao conxunto de transicións a avaliar en cada ciclo (T_a).

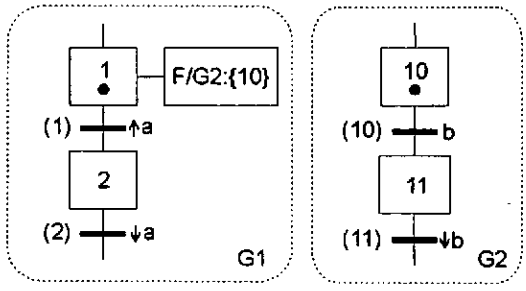


Figura 8.14. Grafcet utilizado como exemplo.

8.6.2. Xestión de temporizacións

Como se explica en (§3.3.3.7), a definición orixinal do Grafcet [1] inclúe un operador para especificar temporizacións nas condicións de transición e nas condicións das asociacións, o que permite modelar accións retardadas e limitadas no tempo. Este operador ten a semántica do elemento de retardo definido no estándar [83] e é considerado, dende o punto de vista da definición do Grafcet, como unha función operativa asociada ao modelo e non como un elemento constitutivo do mesmo [30]. En (§6.3) coméntanse os aspectos a considerar cando se modifica o operador anterior para permitir a utilización de expresións C++ na súa condición. Neste apartado detállanse os aspectos relacionados coa implementación da semántica do operador de temporización e co manexo das temporizacións durante a evolución do modelo.

As temporizacións implementadas no intérprete Grafcet teñen dúas posíbeis semánticas, como mostra a Figura 8.15. A semántica utilizada durante a interpretación indícase como parámetro do método *playModel* (liña 2807) cando esta é iniciada. A diferencia entre ambas semánticas está no comportamento do valor do temporizador cando está activado e a súa condición deixa de ser certa. Nesas circunstancias, se a condición de activación do temporizador volve a ser certa antes de que transcorra o tempo de desactivación t_2, poden darse dúas posibilidades:

1. Que o temporizador se desactive cando transcorra o tempo t_2. Esta semántica correspóndese coa dun temporizador non redisparábel.
2. Que o temporizador non se desactive cando transcorra o tempo t_2. Esta semántica correspóndese coa dun temporizador redisparábel.

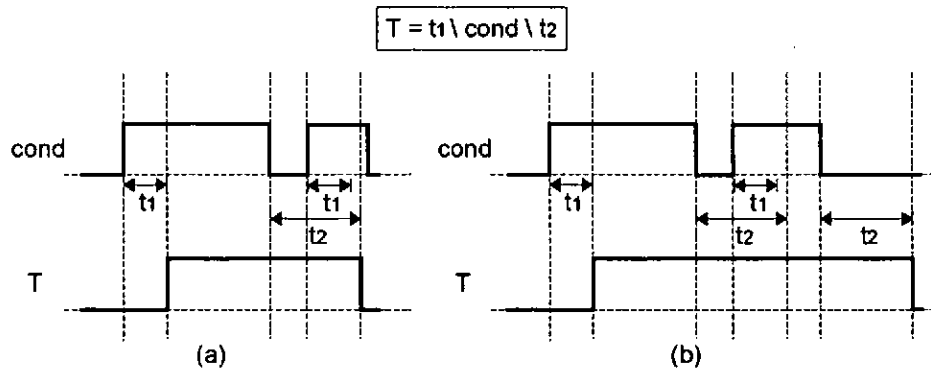


Figura 8.15. Semántica dos temporizadores: a) non redisparábel; e b) redisparábel.

O seguinte pseudocódigo mostra unha versión simplificada da implementación do método que realiza os cálculos precisos para a xestión dos temporizadores. Este método é executado polo algoritmo de interpretación (Figura 8.8) ao inicio de cada ciclo do intérprete, na escala de tempo externa. En consecuencia os cambios de estado dos temporizadores son recibidos polo intérprete xunto cos eventos do proceso.

```
3185. void DefaultRunningPolicy::calculate_timers(GrafcetGame* game,
3186.                                              IVMServices* vmachine)
3187. {
3188.    // avaliar temporizadores activos
3189.    active_timers = game->getActiveTimers();
3190.    for each timer (t) in active_timers
3191.       t.evaluate(this);
3192.    end for
3193.    // almacenar novos eventos
3194.    game->updateTimerEvents();
```

```
3195.    // activar/desactivar temporizadores da máquina virtual
3196.    vmachine->doSchedules();
3197.    vmachine->doUnschedules();
3198. }
```

Para reducir o número de condicións de temporización que é preciso avaliar —obtidas mediante o método *getActiveTimers* do xogo Grafcet (liña 2883)— utilizouse a mesma técnica que se explicou no apartado anterior para as receptividades, de xeito que en cada ciclo son avaliadas as condicións de temporización que conteñan variábeis non booleanas e as que conteñan unicamente variábeis booleanas afectadas por algún dos eventos considerados.

Para o manexo das condicións de temporización do modelo Grafcet a interpretar, o xogo Grafcet almacena unha instancia da clase *TimerScheduleInfo* para cada temporizador —estas instancias son manexadas na implementación da clase *TimerInfo* (Figura 8.12)—. O código da declaración simplificada desta clase é o seguinte:

```
3199. class TimerScheduleInfo
3200. {
3201. private:
3202.   string timer_name; // nome do temporizador no modelo
3203.   bool retrigger;    // ¿é redisparábel?
3204.   RTTimerInfo* info; // información proporcionada polo compilador
3205.   bool last_eval;    // resultado da última avaliación
3206. private:
3207.   // métodos auxiliares
3208.   bool evaluateCondition(RunningPolicy* env);
3209.   void setTimerVar(bool value, RunningPolicy* env);
3210.   void startT1(RunningPolicy* env);
3211.   void stopT1(RunningPolicy* env);
3212.   void startT2(RunningPolicy* env);
3213.   void stopT2(RunningPolicy* env);
3214. public:
3215.   // manexo da temporización
3216.   void evaluate(RunningPolicy* env);
3217. };
```

Como pode verse, esta clase declara os atributos e métodos auxiliares necesarios para implementar no método *evaluate* (liña 3216) o control da temporización, que se corresponde co diagrama de estados da Figura 8.16. Nótese que os estados WAIT e T1T2_COUNT son utilizados unicamente nos temporizadores con semántica redisparábel. O significado dos eventos e accións utilizados nas transicións do diagrama son os seguintes:

- *up(cond) e down(cond)*, indican que a condición do temporizador pasa a ser certa ou falsa, respectivamente. Estes eventos son detectados internamente no método *evaluate* da clase *TimerScheduleInfo*. Esta clase almacena no atributo *last_eval* (liña 3205) o valor do resultado da última avaliación da condición e o compara co resultado actual, devolto polo método privado *evaluateCondition* (liña 3208). Este método executa a función do modelo que contén o código C++ da condición; a súa implementación é semellante á do método *evaluate* (liñas 3096-3102) explicado anteriormente para as receptividades.
- *up(t_1) e up(t_2)*, representan notificacións asíncronas (§7.1.2.4.2) recibidas dende o núcleo da máquina virtual cando transcorren os tempos t_1 e t_2 respectivamente. Ao recibir unha destas notificacións o temporizador é almacenado na lista de temporizadores activos —os devoltos polo método *getActiveTimers* (liña 2883)— do xogo Grafcet, para ser avaliado no seguinte ciclo de execución do intérprete. A medición dos tempos t_1 e t_2 realízase utilizando os servicios de temporización da máquina virtual (§7.3.2), a través dos métodos privados *startT1* (liña 3210), *stopT1* (liña 3211), *startT2* (liña 3212) e *stopT2* (liña 3213).

- A activación e desactivación da variábel que almacena o valor do temporizador no modelo é indicado no diagrama mediante as accións *enable* e *disable*, respectivamente. Ambas accións son realizadas utilizando o método privado *setTimerVar* (liña 3209).

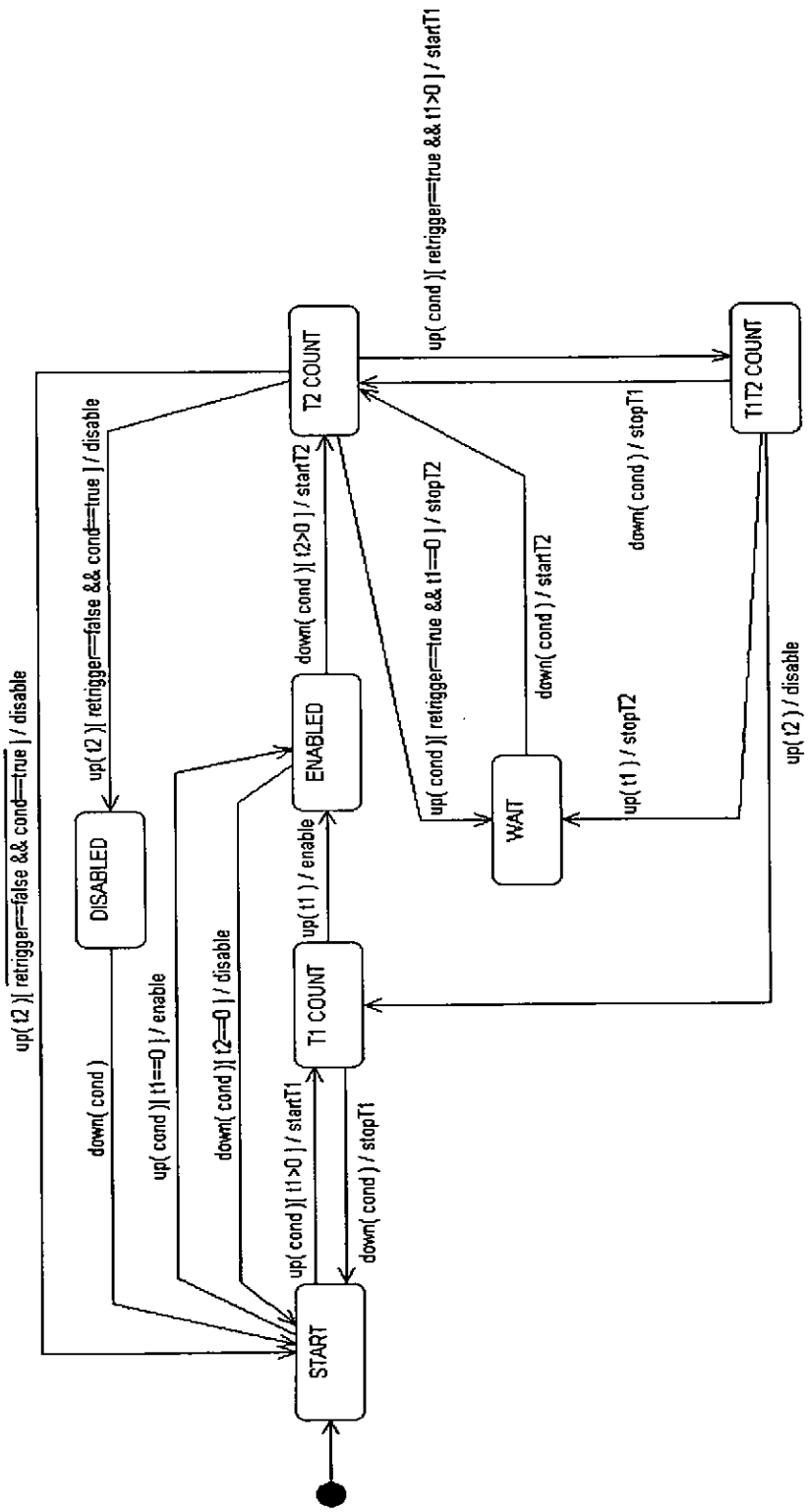


Figura 8.16. Diagrama de estados dun temporizador Grafcet.

A Figura 8.18 mostra un exemplo da secuencia de mensaxes intercambiadas na colaboración entre a política de execución, o xogo Grafcet, os servicios de temporización do núcleo da máquina virtual e a instancia da clase *TimerScheduleInfo* que xestiona o estado do temporizador. Nótese que debido á natureza cíclica do intérprete e a que, como se indicou anteriormente, os temporizadores son avaliados ao inicio de cada ciclo, existen unhas latencias asociadas á planificación dos temporizadores na máquina virtual e ao tempo de ciclo do intérprete. Estas latencias son variábeis e producen unha perda de precisión nas temporizacións, o que pode non ser aceptábel en procesos con requisitos temporais moi estrictos.

A existencia destas latencias obriga a modificar o diagrama de estados da Figura 8.16 para considerar a posibilidade de que se produza algún evento durante a latencia de resposta. Na Figura 8.17 mostrase un exemplo da diferencia entre un caso ideal e un caso con latencia. No caso ideal, a notificación do cumprimento do tempo t_1 é procesada instantaneamente e, polo tanto, o valor da condición será *true*. Sen embargo no caso con latencia, dende que se recibe a notificación ata que se procesa, transcorre un tempo no que o valor da condición pode deixar de ser certo.

O diagrama de estados no que se teñen en conta as latencias móstrase na Figura 8.19. Cada estado no que hai un temporizador activo estará agora formado por varios subestados que representan as diferentes posibilidades que introducen os tempos de latencia. Por exemplo no caso da Figura 8.17 o temporizador está no estado T1_COUNT:T1COUNTING, cando se recibe a notificación de ter transcorrido o tempo t_1 —evento *up(t_1)*— o temporizador pasa ao estado T1_COUNT:T1_EXPIRED, no que se mantén ata ser avaliado no seguinte ciclo do intérprete[79]. No momento da avaliación compróbase o resultado actual da condición, se este segue sendo certo, o temporizador pasa ao estado ENABLED igual que no comportamento ideal, en caso contrario pasa ao estado START ou T2_COUNT:T2COUNTING dependendo do valor de t_2. Nótese que ao avaliar o temporizador no ciclo seguinte ao de recibir a notificación, non existe a posibilidade de que se produza máis dun cambio no valor da súa condición xa que só se obteñen novos valores para as variábeis usadas na condición ao comezo de cada ciclo.

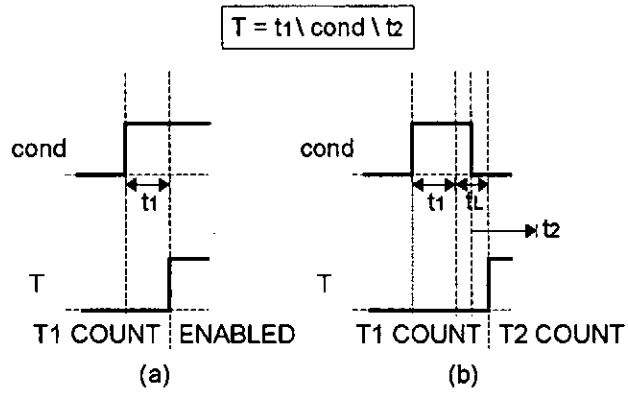


Figura 8.17. Efecto da latencia de resposta no control dun temporizador: a) comportamento ideal; e b) comportamento con latencia.

[79] Para reducir a información das transicións do diagrama, omitiuse o nome do evento naquelas que son avaliadas ao executarse o método *evaluate* do temporizador, e unicamente se indica nelas a condición de transición.

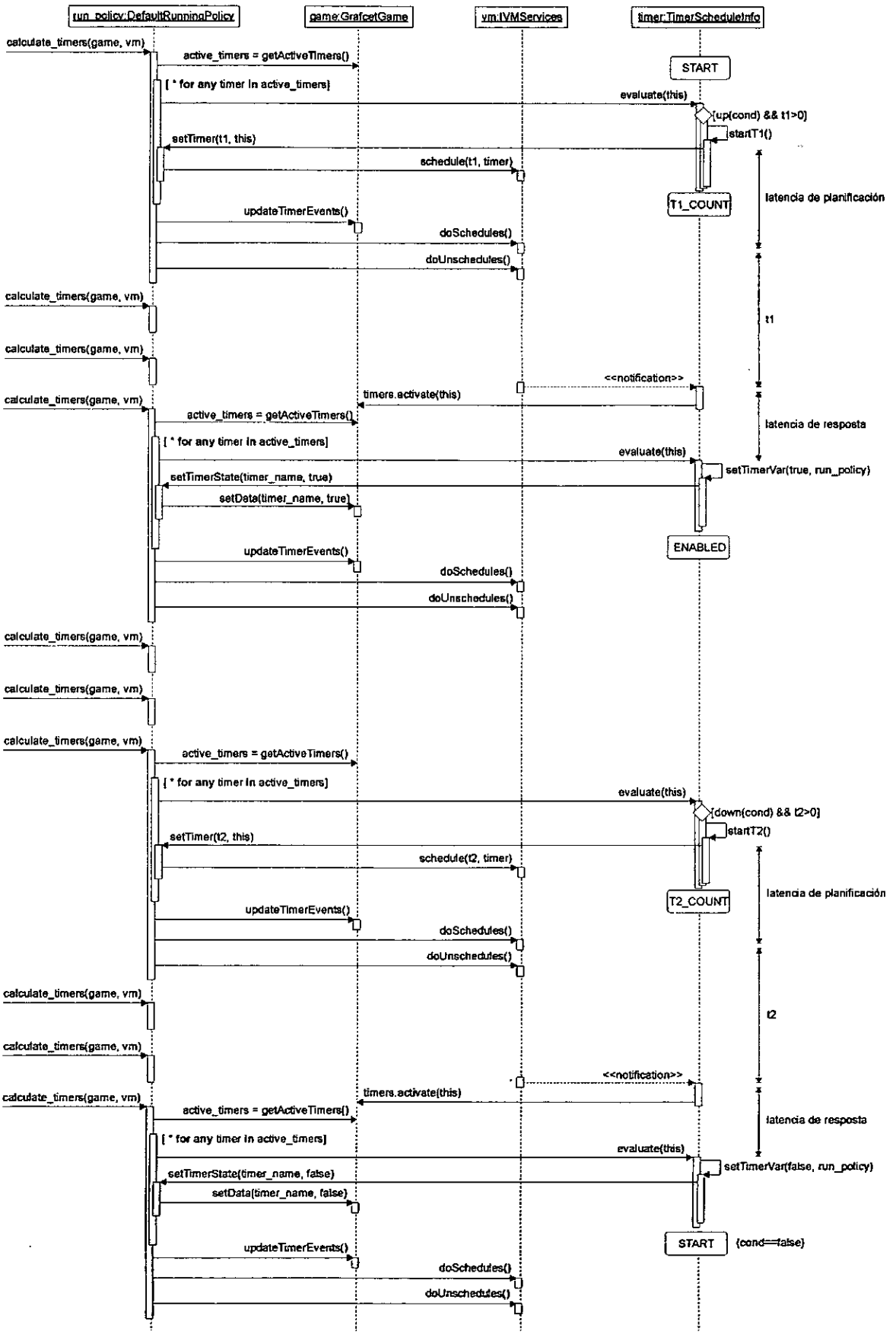


Figura 8.18. Secuencia de mensaxes para a xestión das temporizacións Grafcet.

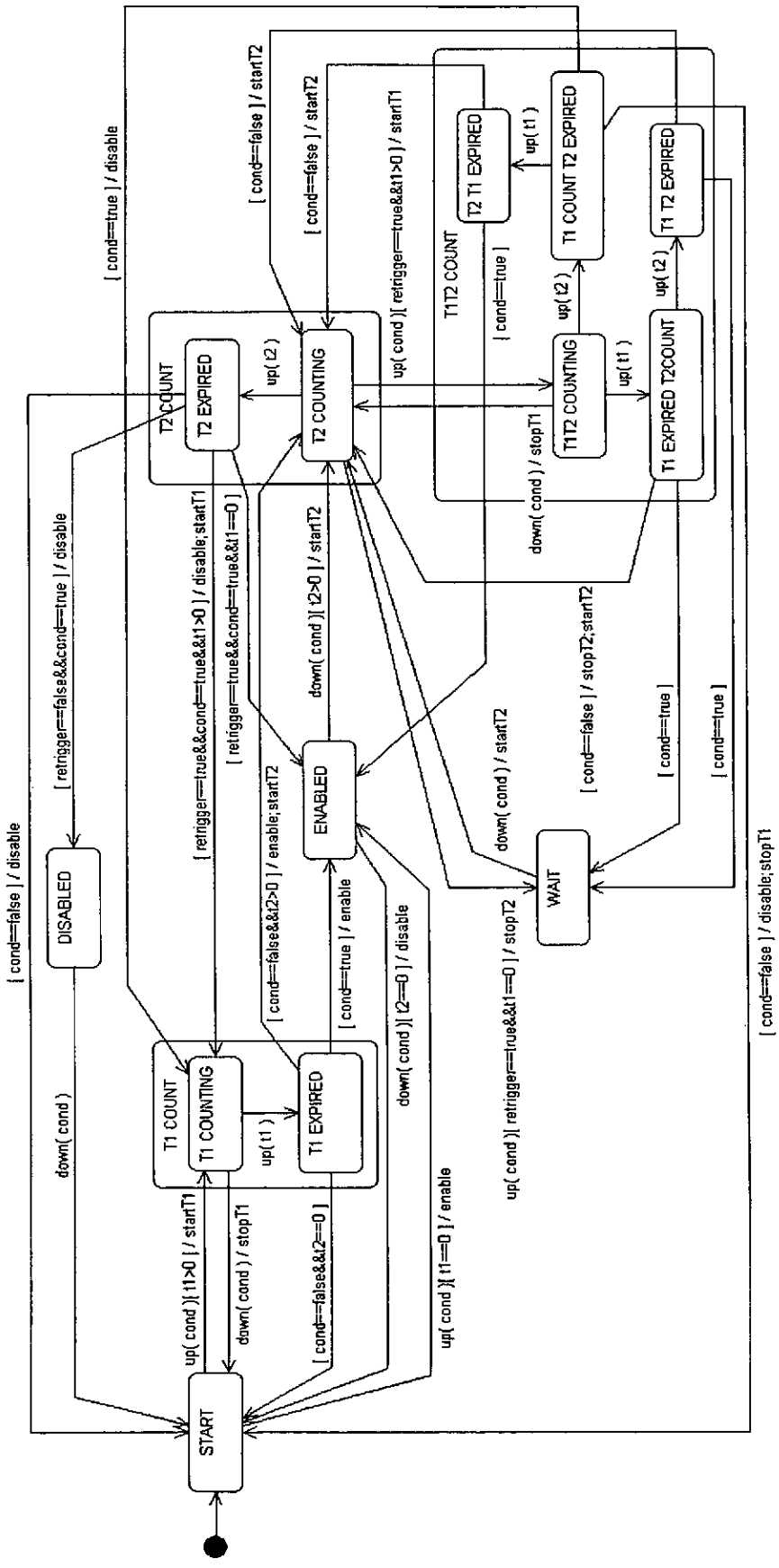


Figura 8.19. Diagrama de estados dun temporizador Grafcet con latencias de resposta.

8.6.3. Execución de accións

En (§5.1.2.3) descríbese a representación de accións e asociacións no metamodelo proposto para o Grafcet. O tipo de asociacións que poden representarse utilizando ese metamodelo é máis amplo que o definido polo estándar IEC 61131-3 [86]. Os tipos básicos de asociacións definidos son: N, R, S, P, P0 e P1; a súa semántica temporal é a descrita en (§3.3.3). Calquera destes tipos básicos pode ter asociada unha condición (C), unha demora (D) e un límite temporal (L). Ademais as accións R, S, P, P0 e P1 poden ser internas ou externas, sendo executadas respectivamente nas evolucións internas do modelo ou unicamente nas externas.

Como se explica en (§3.6.1.3), o estándar IEC 61131-3 define un bloque de control de accións que describe a semántica utilizada para a execución das accións nos SFCs. O equivalente a este bloque é implementado no intérprete Grafcet mediante a clase *ActionScheduleInfo*. Durante a interpretación do modelo almacénase unha instancia desta clase para cada acción no xogo Grafcet —estas instancias son manexadas na implementación da clase *ActionInfo* (Figura 8.12)—. O código simplificado da súa declaración é o seguinte:

```
3218. // Tipos básicos de asociacións
3219. enum ActionType {N, R, S, P, P1, P0};
3220.
3221. class ActionScheduleInfo
3222. {
3223. private:
3224.   string action_name;       // nome da acción no modelo
3225.   bool state;               // estado da acción
3226.   RTActionType code_type;   // tipo de acción (§5.1.2.1)
3227.   bool flags[6];            // 1 "flag" por cada tipo básico de asociación
3228.   unsigned count[3];        // número de asociacións N, R e S activas
3229.   bool stored;              // ¿valor da acción almacenado?
3230. private:
3231.   void runAction(RunningPolicy* env);
3232.   void setActionValue(bool value, RunningPolicy* env);
3233. public:
3234.   void set(ActionType type);
3235.   void reset(ActionType type);
3236.   void run(RunningPolicy* env);
3237. };
```

Ademais da información sobre o nome da acción no modelo, o tipo de acción e o seu estado de activación, cada instancia da clase *ActionScheduleInfo* almacena nos atributos *count* (liña 3228) e *flags* (liña 3227) o número e tipo das asociacións activas que afecten ao valor do seu estado nunha situación do modelo. Os métodos *set* (liña 3234) e *reset* (liña 3235) permiten modificar esta información. O valor da variábel do modelo que almacena o estado de activación da acción é modificado mediante o método privado *setActionValue* (liña 3232), e o método *runAction* (liña 3231), cunha implementación semellante á do método *evaluate* (liñas 3096-3102) explicado anteriormente para as receptividades, executa a función do modelo que contén o código C++ da acción. O control da execución da acción é realizado no método *run* (liña 3236), que ten a seguinte implementación:

```
3238. void ActionScheduleInfo::run(RunningPolicy* env)
3239. {
3240.   // calcular estado da acción
3241.   if (flags[R])
3242.     state = stored = false;
3243.   else if (flags[N] || flags[P] || flags[P1] || flags[P0] || flags[S]
3244.     state = true;
3245.     if (flags[S])
3246.       stored = true;
3247.     end if
```

```
3248.  else if (!stored)
3249.    state = false;
3250.  end if
3251.  // activar/desactivar valor da acción no modelo
3252.  setActionValue(state, env);
3253.  // executar acción
3254.  if (state)
3255.    runAction(env);
3256.  end if
3257.  // desactivar "flags" accións impulsionais
3258.  if (flags[P] || flags[P1] || flags[P0])
3259.    flags[P] = flags[P1] = flags[P0] = false;
3260.  end if
3261.}
```

O cálculo do estado de activación da acción faise nas liñas 3241-3250 considerando os tipos das asociacións activas. As asociacións tipo R teñen prioridade sobre as demais, polo que sempre que haxa unha asociación tipo R activa o valor da acción será desactivado (liña 3242) independentemente da existencia doutras asociacións activas. En caso de non haber ningunha asociación tipo R activa e haber algunha doutro tipo, o valor da acción é activado (liña 3244). Se ademais a asociación é de tipo S actívase o valor do atributo *stored* (liña 3246), para indicar que o valor que ten a acción é almacenado. No caso de non haber ningunha asociación activa, o valor da acción desactívase se non é un valor almacenado (liña 3249). Nótese que se o valor de activación da acción é almacenado, unicamente pode ser desactivado por unha asociación tipo R. En caso de non ser almacenado desactívase simplemente coa ausencia dunha asociación activa. Unha vez calculado o estado de activación da acción actualízase o seu valor no modelo (liña 3252) e execútase o código da acción en caso de estar esta activa (liña 3255). Por último desactívanse os indicadores correspondentes ás asociacións impulsionais activas, de xeito que afecten á execución da acción unicamente unha vez.

O método da política de execución que executa as accións do modelo realiza, para unha situación dada, dúas operacións diferentes: primeiro avalía as asociacións activas e despois calcula o estado de activación das accións e executa as activas. A versión simplificada do seu pseudocódigo é a seguinte:

```
3262.void DefaultRunningPolicy::run(GrafcetGame* game,
3263.                               IVMServices* vmachine,
3264.                               TimeScale time_scale)
3265.{
3266.   // avaliación das asociacións activas e das que deixan de estalo
3267.   active_associations,
3268.    disabled_associations = game->getActionAssociations(time_scale)
3269.   for each association (a) in active_associations
3270.     a.evaluate(this)
3271.   end for
3272.   for each association (d) in disabled_associations
3273.     a.onDeactivate(this)
3274.   end for
3275.   // execución de accións activas
3276.   actions = game->getActions()
3277.   for each action (act) in actions
3278.     act.run(this)
3279.   end for
3280.   // sincronización da "caché"
3281.   flushCache()
3282.}
```

Nótese que o mesmo método é utilizado tanto nas evolucións internas como nas externas do modelo, os aspectos específicos de cada escala son mantidos no xogo Grafcet. O número de

asociacións a avaliar en cada ciclo redúcese utilizando coas asociacións condicionais a mesma técnica que coas receptividades e temporizadores, de xeito que as que conteñan unicamente variábeis booleanas non afectadas por algún dos eventos considerados non son avaliadas. Unha vez executadas as accións actualízanse os datos do modelo e, se é preciso, os valores das saídas do proceso mediante o método privado *flushCache* (liña 3281), que é explicado en (§8.6.4).

Para a avaliación e manexo das asociacións, o xogo Grafcet almacena unha instancia da clase *AssociationScheduleInfo* para cada asociación do modelo a interpretar —estas instancias son manexadas na implementación da clase *AssociationInfo* (Figura 8.12)—. O código da declaración simplificada desta clase é o seguinte:

```
3283. class AssociationScheduleInfo
3284. {
3285. private:
3286.   bool last_eval;              // resultado da última avaliación
3287.   ActionScheduleInfo* action;  // acción modificada pola asociación
3288.   RTActionInfo* info;          // información proporcionada polo compilador
3289. private:
3290.   bool evaluateCondition(RunningPolicy* env);
3291. public:
3292.   void evaluate(RunningPolicy* env);
3293.   void onDeactivate(RunningPolicy* env);
3294. };
```

Cada instancia da clase almacena o resultado da última avaliación da condición da asociación, un apuntador á instancia da clase *ActionScheduleInfo* que contén a información da acción cuxo estado de activación é modificado pola asociación, e un apuntador á información da asociación xerada polo compilador (§6.4). En canto aos métodos da clase, o método privado *evaluateCondition* (liña 3290), cunha implementación semellante á do método *evaluate* (liñas 3096-3102) explicado anteriormente para as receptividades, executa a función do modelo que contén o código C++ da condición da asociación; e os métodos *evaluate* (liña 3292) e *onDeactivate* (liña 3293) implementan o control da asociación.

A Figura 8.20 mostra a representación dunha asociación xenérica e as relacións temporais existentes entre o tempo de activación da etapa á que está asociada e o tempo de activación da propia asociación tendo en conta os posíbeis tempos de demora (T_D) e límite (T_L) especificados no modelo. Se non se especifican tempos de demora e límite o tempo de activación da asociación coincide co da etapa. Nótese que os tempos de demora e límite aparecen especificados na condición tal e como serían representados polo compilador Grafcet (§6.5.1.2).

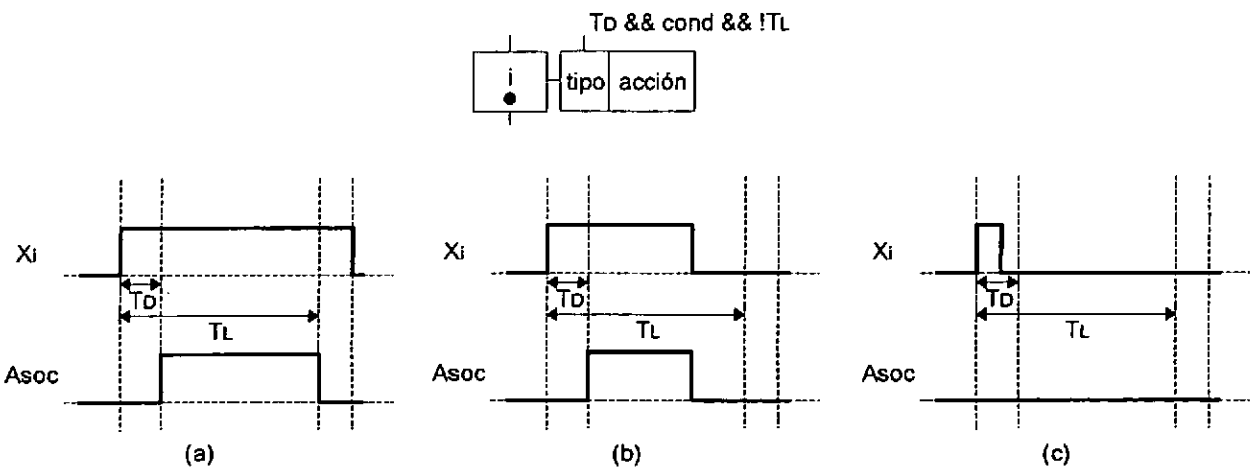


Figura 8.20. Relación temporal entre o tempo de activación dunha etapa e o dunha asociación asociada a ela: a) TD ≤ TL ≤ T(Xi); b) TD ≤ T(Xi) ≤ TL; e c) T(Xi) ≤ TD ≤ TL.

O método *evaluate* (liña 3292) é executado en cada ciclo do intérprete mentres a asociación estea activa, e o método *onDeactivate* (liña 3293) no ciclo no que deixa de estalo. A implementación destes métodos activa e desactiva —en función do valor da condición calculado co método privado *evaluateCondition* (liña 3290)— o "flag" correspondente ao tipo da asociación na instancia da clase *ActionScheduleInfo* referenciada polo atributo *action* (liña 3287). A semántica temporal do "flag" de cada tipo de asociación é mostrada na Figura 8.21 en función dos cambios na condición da asociación[80]. A activación e desactivación do "flag" faise, respectivamente, mediante os métodos *set* (liña 3234) e *reset* (liña 3235) da clase *ActionScheduleInfo*, aos que se lles pasa como parámetro o tipo da asociación.

Nótese que a semántica de activación dos "flags" dos tipos N, R e S é coincidente. A diferencia entre elas estará polo tanto na forma en que son manexadas as activacións dos seus "flags" no método *run* da clase *ActionScheduleInfo* (liñas 3241-3250). No que respecta ás asociacións impulsionais, o "flag" das tipo P1 é activado a primeira vez que a condición é certa, xa sexa no instante de activarse a asociación (Figura 8.21) ou unha vez activada (Figura 8.22). O das tipo P é activado cada vez que a condición se volve certa (Figura 8.22), incluído o caso no que xa sexa certa no instante de activarse a asociación (Figura 8.21). O das tipo P0 é activado no instante en que a asociación deixe de estar activa se a condición é certa (Figura 8.21). En caso de non selo non seria activada (Figura 8.22). Os "flags" de todos os tipos de accións impulsionais son desactivados no método *run* da clase *ActionScheduleInfo* (liñas 3258-3260) para garantir que a acción só é executada unha vez en cada activación.

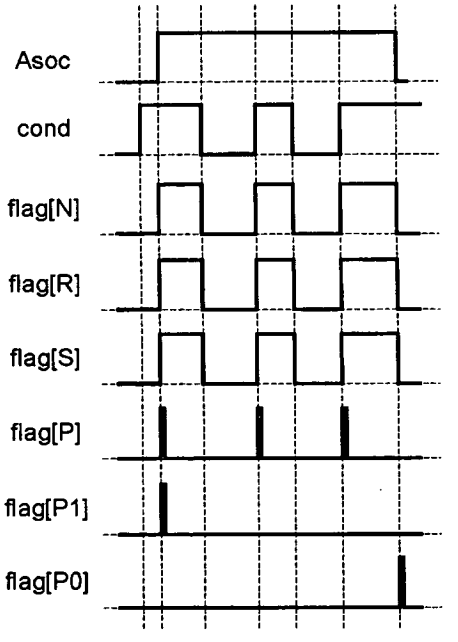


Figura 8.21. Semántica temporal dos "flags" dos diferentes tipos de asociacións.

[80] Coa semántica aquí descrita é posíbel representar todos os tipos de asociacions definidos no estándar IEC 61131-3, coa excepción dos tipos SD e SL que non teñen equivalente no intérprete Grafcet.

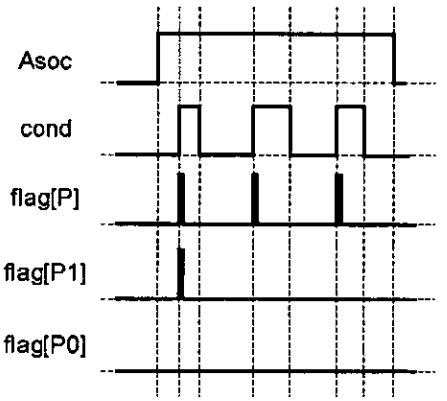


Figura 8.22. Semántica temporal dos "flags" das asociacións impulsionais.

8.6.4. Consulta e modificación de variábeis

Os métodos *getModelVarEvent* (liña 3070), *getSystemVarEvent* (liña 3071) e *getStepState* (liña 3072) da clase *RunningPolicy* invocan directamente aos métodos de igual nome no xogo Grafcet (§8.5.1.4), permitindo consultar a ocorrencia de eventos e o estado de activación das etapas do modelo. De igual modo o método *getTimerState* (liña 3073) consulta o estado dun temporizador invocando o método *getData* (liña 2869) do xogo Grafcet. Os métodos anteriores utilízanse na execución das funcións que conteñen o código C++ das condicións e accións xeradas polo compilador Grafcet. Estes métodos unicamente consultan valores, sen que exista a posibilidade de modificar dende estas funcións o estado dunha etapa, temporizador ou evento.

Sen embargo os métodos *getModelVar* (liña 3075) e *getSystemVar* (liña 3079) consultan valores de variábeis que poden ser modificados dende as funcións C++ utilizando os métodos *setModelVar* (liña 3077) e *setSystemVar* (liña 3081), segundo se explica en (§6.3.3.3). En cada ciclo de execución do intérprete a avaliación de todas as condicións e a execución de todas as accións teñen que realizarse utilizando os valores que as variábeis do modelo e do proceso teñan ao comezo do ciclo. Como a execución de accións pode modificar estes valores é preciso implementar algún mecanismo que manteña os valores que as variábeis teñen ao inicio do ciclo, almacene as modificacións e actualice os valores modificados ao final do ciclo. Este problema é resolto implementando na clase *RunningPolicy* unha "caché" de datos. Os elementos almacenados nesta "caché" son instancias da clase parametrizada *CacheEntry*, cuxa declaración simplificada é a seguinte:

```
3295. template <class T>
3296.   class CacheEntry : public ptrSeqElem
3297. {
3298. private:
3299.   string name;              // identificador da variábel
3300.   T initial_value;          // valor ao inicio do ciclo
3301.   T modified_value;         // valor modificado
3302.   bool modified, updated; // indicadores auxiliares
3303. public:
3304.   const string& getKey(void) const;
3305.   bool isUpdated(void) const;
3306.   bool isModified(void) const;
3307.   // acceso e modificación do valor
3308.   template <class T> void getValue(T& value);
3309.   template <class T> void setValue(const T& value);
3310.   // actualización e sincronización do valor
3311.   void update();
3312.   void flush(RunningPolicy& run_policy);
3313. };
```

Cada entrada da "caché" almacena dous valores, o que a variábel tiña ao comezo do ciclo actual —atributo *initial_value* (liña 3300)— e o valor modificado durante o ciclo —atributo *modified_value* (liña 3301)—. O método *getValue* (liña 3308) devolve o valor de comezo de ciclo, mentres que o método *setValue* (liña 3309) almacena o valor modificado. Este método tamén detecta o intento de modificar a variábel con valores diferentes máis dunha vez na mesma evolución do modelo (§3.3.3.8), nese caso indícase o erro utilizando os servicios do núcleo da máquina virtual. Para realizar a actualización dos valores na "caché" e a sincronización da "caché" cos valores do modelo, utilízanse dous indicadores booleanos e os métodos *update* (liña 3311) e *flush* (liña 3312). A "caché" é representada mediante a clase *DataCache*, que proporciona os métodos que permiten xestionar un conxunto de instancias da clase *CacheEntry*. A declaración simplificada da súa interface pública é a seguinte:

```
3314. class DataCache
3315. {
3316. public:
3317.   bool empty() const;
3318.   bool modified() const;
3319.   void removeAll();
3320.   // consulta e modificación das entradas da "caché"
3321.   template <class T> bool insertValue(string varID, const T& value);
3322.   template <class T> bool getValue(string varID, T& value);
3323.   template <class T> bool setValue(string varID, const T& value);
3324.
3325.   // actualización e sincronización do valor
3326.   void update();
3327.   void flush(RunningPolicy& run_policy);
3328. };
```

Os métodos desta interface permiten consultar se hai valores na "caché" —método *empty* (liña 3317)—, se algún dos valores foi modificado —método *modified* (liña 3318)—, baleirar os seus contidos —método *removeAll* (liña 3319)—, actualizalos —método *update* (liña 3326)— e sincronizalos cos valores do modelo —método *flush* (liña 3327)—. A inserción, consulta e modificación das entradas almacenadas na "caché" realízase cos métodos *insertValue* (liña 3321), *getValue* (liña 3322) e *setValue* (liña 3323), respectivamente.

O funcionamento da "caché" é o seguinte: ao comezo de cada ciclo de execución do intérprete, a "caché" está baleira. Cada invocación dos métodos *getModelVar* e *getSystemVar* da clase *RunningPolicy* dende unha acción ou condición realiza primeiro unha busca do valor na "caché". Se o valor non está na "caché" crease unha nova entrada co valor obtido do xogo Grafcet ou da base de datos de E/S da máquina virtual (§7.3.1), dependendo do caso. Do mesmo xeito, os métodos *setModelVar* e *setSystemVar* da clase *RunningPolicy* tentan modificar o valor existente na "caché". En caso de que o valor a modificar non estea na "caché", crease unha nova entrada co novo valor. A Figura 8.23 mostra como exemplo as secuencias de mensaxes das colaboracións que implementan os métodos *getSystemVar* e *setSystemVar*.

A actualización dos valores da "caché" e a sincronización cos valores do modelo faise mediante o método privado *flushCache* da clase *DefaultRunningPolicy*, que é invocado ao final do método *run* (liña 3281) unha vez executadas as accións activas do modelo. A actualización consiste en asignar o valor modificado ao valor inicial na entrada da "caché" para poder reutilizala, mentres que a sincronización consiste en actualizar no xogo Grafcet os valores modificados na "caché". O instante no que se realiza cada unha destas operacións depende do tipo de variábel almacenada na "caché":

1. Os valores das variábeis do proceso son actualizadas na "caché" despois de cada evolución interna do modelo e sincronizadas co xogo Grafcet despois de cada evolución externa.

Nótese que unicamente as saídas do proceso poden verse afectadas por unha actualización ou sincronización, xa que o valor das entradas non pode ser modificado pola avaliación de condicións e a execución de accións.

2. As variábeis internas do modelo son actualizadas na "caché" e sincronizadas co xogo Grafcet despois de cada evolución, xa sexa interna ou externa. A escala de tempo actualizada no xogo Grafcet será a mesma na que se produza a evolución do modelo.

Todas as entradas da "caché" son eliminadas despois dunha evolución externa do modelo. Os valores das saídas do proceso modificadas na evolución son actualizadas na base de datos de E/S segundo se explica en (§8.4.4).

Unha consecuencia derivada da utilización da "caché" é a reducción dos tempos de acceso ás variábeis do proceso almacenadas na base de datos de E/S. A base de datos é executada nun proceso da máquina virtual diferente ao do intérprete Grafcet, polo que é preciso utilizar algún mecanismo de exclusión mutua no acceso aos valores da base de datos de E/S coa correspondente penalización temporal que isto implica. Coa utilización da "caché", unicamente se accede á base de datos a primeira vez que se consulte ou modifique o valor dunha variábel do proceso en cada ciclo do intérprete. En consecuencia cando un mesmo valor é accedido en múltiples ocasións en cada ciclo do intérprete, xa sexa porque é utilizado en diferentes condicións ou accións ou en diferentes evolucións internas do modelo, conséguese unha reducción significativa do tempo preciso para a execución de condicións e accións.

8.7. Conclusións

Neste capítulo describiuse a arquitectura e o funcionamento do intérprete utilizado para a interpretación dos modelos Grafcet definidos polo usuario. O intérprete foi deseñado para que os módulos da súa arquitectura podan ser substituídos dinamicamente e facilitar así a implementación de novas características e algoritmos sen que sexa preciso recompilar o intérprete. Tamén se implementou a xestión da información de dúas escalas de tempo diferentes durante a evolución dos modelos, de xeito que podan utilizarse algoritmos de evolución con ou sen busca da estabilidade. Ademais poden configurarse mediante parámetros diferentes opcións relacionadas coa utilización das dúas escalas de tempo.

A arquitectura do intérprete foi implementada para permitir a escalabilidade. Toda a información precisa para a evolución de cada modelo é xestionada mediante un 'xogo Grafcet', que tamén contén a información utilizada polas técnicas que optimizan a execución do código de condicións e accións do modelo. Isto permite desacoplar a información de interpretación do modelo da lóxica utilizada para a súa evolución e a execución do seu código. Deste xeito resulta simple modificar o intérprete para implementar características máis complexas, como por exemplo a interpretación simultánea de múltiples modelos con diferentes opcións de configuración. Entre as melloras que poden incluírse no intérprete poden citarse as seguintes:

1. A utilización da información sobre os eventos de cada condición do modelo para descartar os non significativos na política de acceso (§8.3.1.1). A recopilación desta información requiriría tamén a modificación do compilador Grafcet.
2. A reducción do tempo de ciclo do intérprete implementando políticas de execución que reduzan os tempos de acceso ás variábeis ou os tempos de execución das condicións e accións, por exemplo distribuíndoas nun sistema multiprocesador.
3. A incorporación dun mecanismo tipo "watchdog" (semellante ao utilizado nos PLCs) que permita especificar o tempo de resposta máximo aceptado polo proceso e configurar a acción a realizar se algún dos ciclos de execución do intérprete supera este límite.

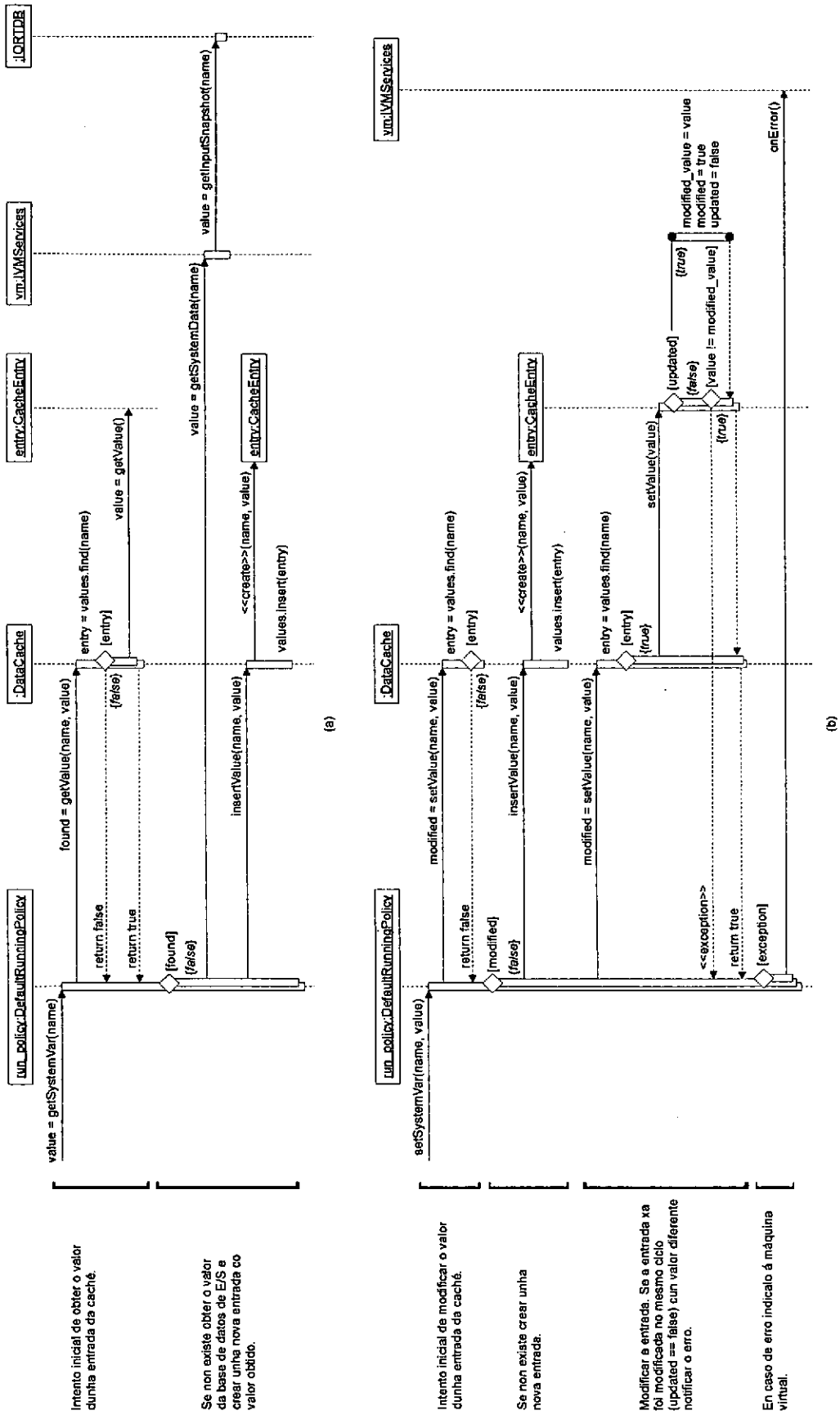


Figura 8.23. Secuencias de mensaxes da implementación dos métodos que acceden aos valores das variábeis do proceso: a) método *getSystemVar*; e b) método *setSystemVar*.

Capítulo 9. Conclusións e futuras liñas de investigación

Neste capítulo resúmense as conclusións extraídas do desenvolvemento da ferramenta proposta, indícanse algunhas melloras a realizar en futuras versións e propóñense futuras liñas de investigación.

9.1. Conclusións

A ferramenta implementada nesta tese de doutoramento proporciona as compoñentes precisas para integrar as descricións de sistemas cunha lóxica secuencial complexa, especificados combinando unha aproximación orientada a obxecto co Grafcet, en ambientes de desenvolvemento de "software" para sistemas industriais nos que se utilice unha aproximación multiformalismo que proporcione asistencia aos enxeñeiros de control durante o ciclo de vida do sistema.

Os modelos Grafcet poden ser integrados en calquera outra ferramenta utilizando o metamodelo Grafcet definido, xa sexa mediante a libraría C++ que o implementa ou incluíndoo na propia ferramenta por outros medios. O código para a execución dos modelos xérase automaticamente utilizando o compilador implementado. A execución e simulación dos modelos realízase na máquina virtual que proporciona o soporte básico para o manexo de eventos, temporizacións e acceso ás magnitudes do proceso, e na que se dispón dun intérprete que pode ser configurado para utilizar diferentes algoritmos de interpretación. A máquina virtual pode ser utilizada en arquitecturas heteroxéneas e, mediante a definición dos "drivers" apropiados, pode interactuarse co proceso mediante múltiples sistemas de E/S diferentes. Ademais esta interacción pode ser configurada para que as magnitudes sexan lidas cada vez que varíen, ou ben a intervalos regulares. A estructuración da aplicación realízase combinando a estructura xerárquica do Grafcet coas propiedades da orientación a obxectos que proporciona a linguaxe C++, que tamén é a linguaxe utilizada (estendida coa agregación de operadores para o manexo de eventos e temporizacións) para programar os contidos das accións e receptividades dos modelos Grafcet, o que permite incluír nos modelos as funcionalidades implementadas en librarías externas, como por exemplo librarías matemáticas, de simulación, de técnicas de "soft-computing", etc. A ferramenta complétase cun editor gráfico que proporciona un punto de acceso común a todas as compoñentes implementadas.

En conxunto as compoñentes da ferramenta permiten que un enxeñeiro de control aplique un proceso de desenvolvemento iterativo baseado na construcción de modelos gráficos executábeis que son validados e refinados progresivamente mediante simulación, e no que é posíbel utilizar

o Grafcet, combinado con outros formalismos, de maneira uniforme ao longo das diferentes fases do ciclo de vida do sistema de control. A combinación dunha aproximación orientada a obxecto cun formalismo gráfico de semántica ben definida para a especificación de secuencias complexas, facilita a estructuración dos modelos nas fases de análise e deseño e a obtención automática do código da aplicación para ser executado en arquitecturas industriais distribuídas.

Durante o deseño e implementación da ferramenta priorizouse a aplicabilidade, flexibilidade e portabilidade das súas compoñentes, deixando nesta primeira versión os aspectos relacionados coa eficiencia na execución e no manexo de memoria nun segundo plano. Isto pode comprobarse na utilización de librarías como STL (baseada no uso de "templates") ou Common C++, que facilitan a portabilidade ao custe dunha menor eficiencia. Tamén na arquitectura da máquina virtual, que está implementada en base á definición de interfaces abstractas e á utilización de conceptos tomados das arquitecturas baseadas en actores [3]. Nestas arquitecturas a comunicación entre os procesos (nomenclatura utilizada na máquina virtual) realízase mediante o intercambio de mensaxes, que son procesados utilizando unha semántica RTC, de xeito que un proceso non ten en conta unha nova mensaxe ata que non remate co procesamento da actual. Esta arquitectura ten tres consecuencias principais:

1. Os procesos teñen un acoplamento débil entre eles, o que facilita a substitución e a reconfiguración dinámica da arquitectura.
2. Elimínanse os problemas derivados da concorrencia no interior dos procesos. Debido a que o procesamento dunha mensaxe non pode ser interrompida, non é precisa a inclusión de mecanismos de exclusión mutua na implementación dos procesos. Nótese que o procesamento dun proceso si pode ser interrompido por outro de maior prioridade, mais isto non afectará ao seu estado interno.
3. Os tempos de resposta ao envío de mensaxes son, en xeral, superiores neste tipo de arquitecturas que nas baseadas no manexo de interrupcións.

Un inconveniente desta arquitectura é o referido ao manexo das temporizacións, que son implementadas como procesos que notifican a finalización dun período de tempo utilizando o mecanismo de paso de mensaxes. Aínda que estas notificacións teñen prioridade sobre as outras mensaxes, non son consideradas ata que o proceso receptor remate o procesamento da mensaxe actual en caso de habela. Isto ten como consecuencia unha perda de precisión que pode non ser aceptábel en certas aplicacións. En consecuencia a aplicabilidade da máquina virtual dependerá basicamente de tres parámetros:

1. A cantidade de memoria dispoñíbel no equipamento de control utilizado.
2. Os tempos de resposta que o proceso a controlar precise.
3. A precisión requirida nas temporizacións.

Ningún dos parámetros anteriores é constante e varían en función do equipamento de control, da configuración da máquina virtual e dos modelos Grafcet concretos que se utilicen. A validación por simulación dos modelos e o axuste das frecuencias de monitorización das magnitudes do proceso son os medios dos que dispón o enxeñeiro de control para comprobar e axustar o funcionamento da aplicación.

En contrapartida a estas limitacións, as arquitecturas implementadas son portábeis e flexíbeis. Así no compilador poden incluírse novas funcionalidades de forma simple engadindo novas fases ou reimplementando as existentes. De igual maneira a implementación da máquina virtual pode ser portada facilmente a diferentes equipamentos de control, poden incluírse múltiples "drivers" para utilizar diferentes dispositivos de E/S, a súa configuración pode ser

modificada dinamicamente, e a interacción do subsistema de aplicación co subsistema de E/S pode ser configurada. As funcionalidades que proporciona a máquina virtual (xestión de eventos e temporizacións, acceso ás magnitudes do proceso, sincronización da imaxe de E/S, etc.) non se limitan á súa utilización co intérprete Grafcet, senón que son comúns no control de DEDS, polo que a máquina virtual pode ser reutilizada na implementación doutras aplicacións, como por exemplo: intérpretes de RdPI ou StateCharts, emuladores de PLC, etc.

Tamén o intérprete Grafcet ten unha arquitectura lóxica flexíbel na que poden substituírse dinamicamente algunha das súas compoñentes, como a política para a obtención de novos eventos, o algoritmo de evolución ou o de execución, por exemplo. A interpretación dos modelos pode realizarse utilizando semánticas con ou sen busca de estabilidade, e nas primeiras, pode configurarse a forma en que os eventos e variábeis son utilizados nas dúas escalas de tempo utilizadas. Aínda que a versión actual só permite a interpretación dun único modelo Grafcet, o intérprete foi deseñado para permitir, con pequenas modificacións, a interpretación simultánea de múltiples modelos. Da mesma maneira, a implementación de novas políticas de evolución facilítase implementando un mecanismo de composición que permite reutilizar as xa existentes.

9.2. Melloras a realizar

Algúns dos aspectos máis relevantes que pode ser mellorados nas diferentes compoñentes da ferramenta presentados nesta documentación son os seguintes:

1. Incluír no metamodelo as extensións adoptadas na última revisión do estándar Grafcet [85] e na máquina virtual un algoritmo de interpretación que implemente as modificacións semánticas debidas a esas extensións.
2. Incluír soporte á utilización dun formato estándar (p.e. XML) para o intercambio de modelos Grafcet entre aplicacións.
3. Incluír soporte á utilización dalgunha linguaxe de descrición de dispositivos (como a utilizada en FieldBus, por exemplo) para representar a información de configuración dos "drivers" de E/S.
4. Implementar na máquina virtual un mecanismo tipo "watchdog" semellante ao dispoñíbel nos PLCs que permita especificar un tempo de resposta máximo para o intérprete e detecte cando este é superado.
5. Adaptación da máquina virtual para permitir a implementación de intérpretes doutros formalismos de modelado de DEDS (RdPI, StateCharts, etc.).
6. Implementación do acceso remoto aos servicios da máquina virtual, actualmente baseado no intercambio de mensaxes, por algún mecanismo de invocación remota como o proporcionado por CORBA, p.e.
7. Optimizar a eficiencia en termos de utilización de memoria e tempo de procesamento das implementacións do compilador e da máquina virtual.
8. Implementación de "drivers" de E/S para ser utilizados con buses de campo (p.e. Profibus ou Fieldbus) e protocolos de intercambio de información con outras aplicacións (p.e. NetDDE ou OPC).

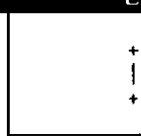
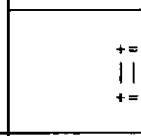
9.3. Futuras liñas de investigación

Algunhas das liñas de investigación que supoñen unha continuación do traballo realizado nesta tese de doutoramento son as seguintes:

1. A análise e proposta de técnicas que permitan tomar en consideración as restricción de tempo e memoria dos equipamentos de control utilizados. A priori isto implica a implementación de versións específicas da máquina virtual máis dependentes do sistema e a modificación do código xerado polo compilador para obter unha versión máis compacta.
2. A análise, proposta e implementación de técnicas para a integración doutras linguaxes e formalismos na especificación de accións e condicións, sendo de especial interese as linguaxes IEC, as linguaxes de fluxo de datos e a utilización de métodos de "soft-computing" (bases de regras, lóxica difusa, etc.).
3. O estudio, proposta e implementación de regras formais de utilización que permitan mellorar a reusabilidade dos modelos Grafcet en ambientes orientados a obxecto mediante mecanismos como a herdanza, a composición ou a parametrización.
4. A análise e proposta de métodos para a integración de formalismos de especificación de sistemas complexos en base á definición dos seus metamodelos.
5. A proposta de métodos para a adaptación de metodoloxías de desenvolvemento "software" á implementación de aplicacións industriais baseadas nos estándares IEC. Ten especial interese a proposta e implementación de extensións a UML, de xeito que poidan aplicarse con ese obxectivo as ferramentas CASE existentes.
6. O estudio, proposta e implementación de extensións á ferramenta orientadas ao desenvolvemento de software para sistemas "batch", baseados no uso de 'receitas'.
7. A análise, proposta e implementación de técnicas para a aplicación da ferramenta no desenvolvemento de aplicacións con arquitecturas distribuídas, como por exemplo as baseadas na utilización de compoñentes ou os sistemas multiaxente.

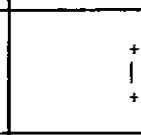
Anexo A. Características do SFC no estándar IEC 61131-3

Neste anexo inclúense as táboas que resumen as características do SFC tal e como son definidas no estándar IEC 61131-3 [86].

Nº	Exemplo	Descrición
1		*Etapa*: representación gráfica con arcos orientados. *** = nome da etapa
		*Etapa inicial*: representación gráfica con arcos orientados. *** = nome da etapa inicial

2	`STEP *** :`
	`  (* Step Body *)`
	`END_STEP`
		Etapa: definición textual sen arcos orientados.
		*** = nome da etapa

	`INITIAL_STEP *** :`
	`  (* Step Body *)`
	`END_STEP`
		Etapa Inicial: definición textual sen arcos orientados.
		*** = nome da etapa inicial

3a	`***.X`	*Flag de estado da etapa* (valor booleano) *** = nome da etapa ***.X = 1 cando *** estea activa, 0 noutro caso
3b		*Flag de estado da etapa* (valor booleano) conexión directa dunha variábel booleana. ***.X ao lado dereito da etapa ***
4	`***.T`	*Tempo de activación da etapa* *** = nome da etapa ***.T = variábel tipo TIME

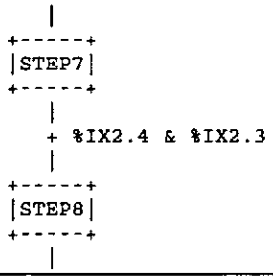
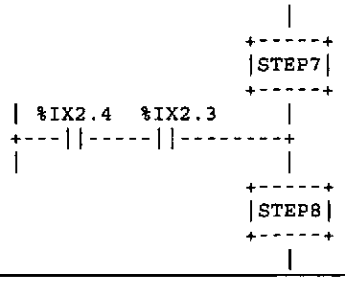
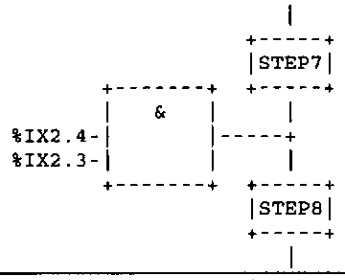
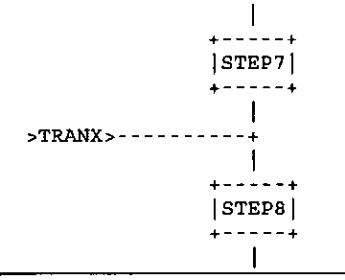
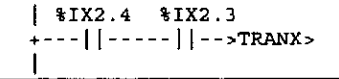
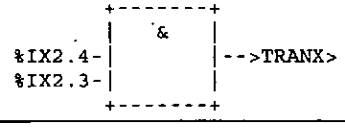
Notas

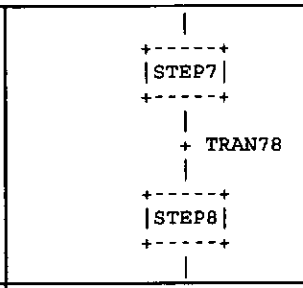
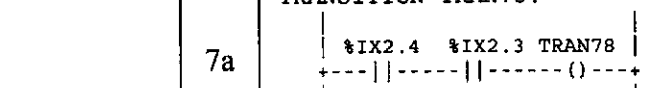
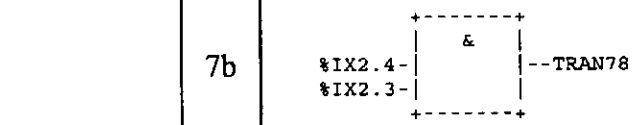
1. Se se inclúe soporte para algunha das características 3a, 3b ou 4, entón debe considerarse un erro o intento de modificación da variábel asociada. Por exemplo, se temos unha etapa chamada S4, as seguintes instruccións ST deberían producir un erro:

```
S4.X := 1; (* ERRO *)
S4.T := t#100ms; (* ERRO *)
```

2. A conexión superior nunha etapa inicial non é precisa cando non teña antecesores.

Táboa A-I. Características das etapas.

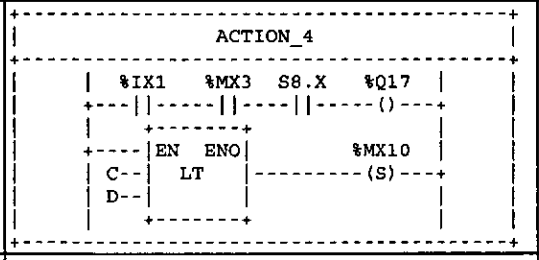
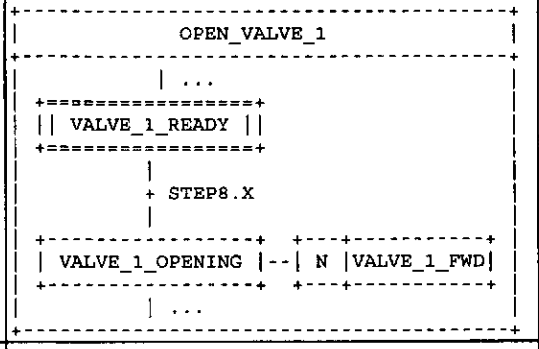
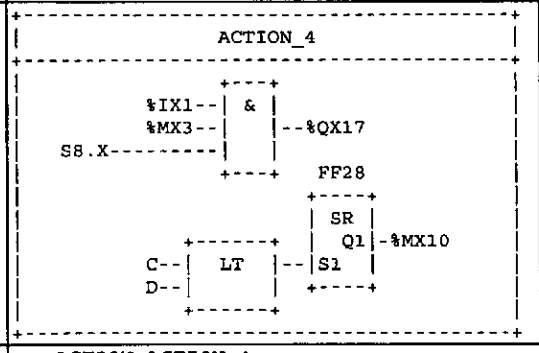
Nº	Exemplo	Descrición
1		Definición da condición de transición mediante ST
2		Definición da condición de transición mediante LD
3		Definición da condición de transición mediante FBD
4		Utilización de conectores nas condicións de transición:
4a		mediante LD
4b		mediante FBD
5	STEP STEP7 : END_STEP TRANSITION FROM STEP7 TO STEP8: := %IX2.4 & %IX2.3; END_TRANSITION STEP STEP8 : END_STEP	Definición da condición de transición mediante ST
6	STEP STEP7 : END_STEP TRANSITION FROM STEP7 TO STEP8: LD %IX2.4 AND %IX2.3 END_TRANSITION STEP STEP8 : END_STEP	Definición da condición de transición mediante IL

7		Utilización do nome de transición nas condicións de transición:
7a	TRANSITION TRAN78:  END_TRANSITION	mediante LD
7b	TRANSITION TRAN78:  END_TRANSITION	mediante FBD
7c	TRANSITION TRANS78 : LD %IX2.4 AND %IX2.3 END_TRANSITION	mediante IL
7d	TRANSITION TRANS78 : := %IX2.4 & %IX2.3; END_TRANSITION	mediante ST

Notas

1. Se se inclúe soporte para a característica 1 da Táboa A-I, entón debe incluírse soporte para cando menos unha das características 1, 2, 3, 4 ou 7 desta táboa.
2. Se se inclúe soporte para a característica 2 da Táboa A-I, entón debe incluírse soporte para cando menos unha das características 5 ou 6 desta táboa.

Táboa A-II. Características das transicións e das condicións de transición.

Nº	Característica	
1	Calquera variábel booleana nun bloque VAR ou VAR_OUTPUT pode ser unha acción	
	Exemplo	**Característica**
2l		Declaración gráfica de bloques de definición de accións mediante LD
2s		Utilización de elementos do SFC nos bloques de definición de accións
2f		Declaración gráfica de bloques de definición de accións mediante FBD

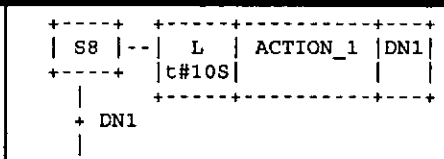
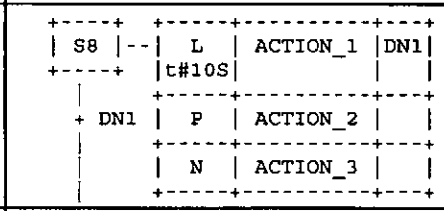
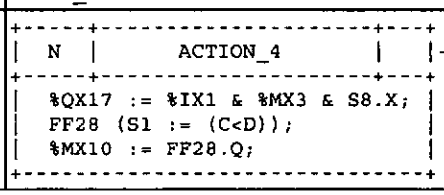
3s	`ACTION ACTION_4:`
	`  %QX17 := %IX1 & %MX3 & S8.X;`
	`  FF28(S1 := (C<D));`
	`  %MX10 := FF28.Q;`
	`END_ACTION`
		Declaración textual de bloques de definición de accións mediante ST

3i	`ACTION ACTION_4:`
	`  LD S8.X`
	`  AND %IX1`
	`  AND %MX3`
	`  ST %QX17`
	`  LD C`
	`  LT D`
	`  S1 FF28`
	`  LD FF28.Q`
	`  ST %MX10`
	`END_ACTION`
		Declaración textual de bloques de definición de accións mediante IL

Notas

1. O flag de estado da etapa S8.X utilízase para obter o valor desexado, que estando S8 desactivado, será %QX17:=0.
2. Se se inclúe soporte para a característica 1 da Táboa A-I, entón tamén debe incluírse para unha ou mais das características desta táboa ou para a característica 4 da Táboa A-IV.
3. Se se inclúe soporte para a característica 2 da Táboa A-I, entón tamén debe incluírse para unha ou mais das características 1, 3s ou 3i desta táboa.

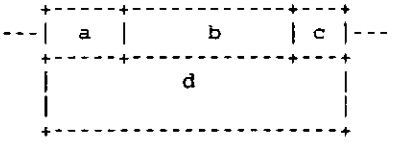
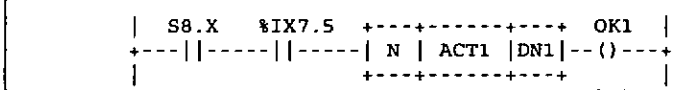
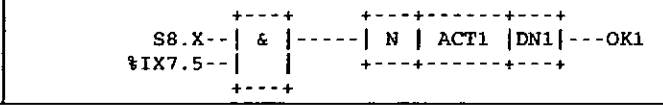
Táboa A-III. Declaración de accións.

Nº	Exemplo	Característica
1		Bloque de definición de accións
2		Bloques de definición de accións concatenados
3	STEP S8: ACTION_1(l, T#10S, DN1) ACTION_2(P) ACTION_3(N) END_STEP	Definición textual das etapas
4		Campo "d" do bloque de definición de accións

Nota

1. Cando se utilice a característica 4, o nome de acción correspondente non pode utilizarse en ningún outro bloque de definición de accións.

Táboa A-IV. Asociación de etapas e accións.

Nº	Característica	Representación Gráfica
1	"a": tipo	
2	"b": nome da acción	
3	"c": indicador booleano	
	"d": definición da acción mediante:	
4	Linguaxe IL	
5	Linguaxe ST	
6	Linguaxe LD	
7	Linguaxe FBD	
	Característica/Exemplo	
8	Utilización de bloques de definición de accións nos diagramas de contactos: 	
9	Utilización de bloques de definición de accións nos diagramas de bloques función: 	

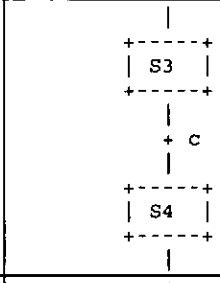
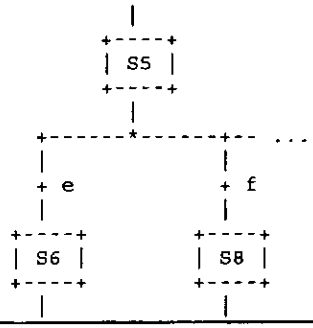
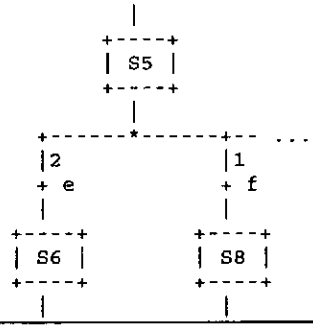
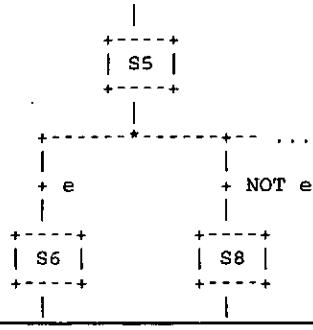
Notas

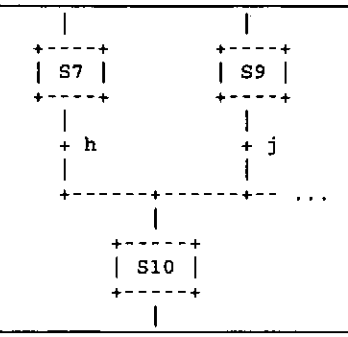
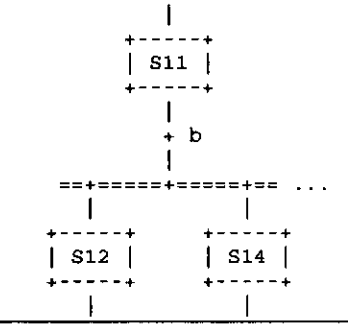
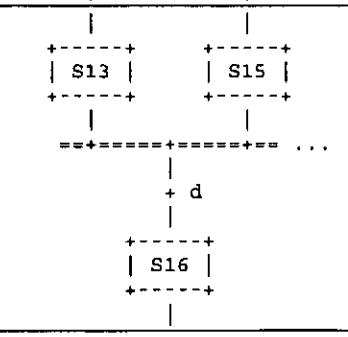
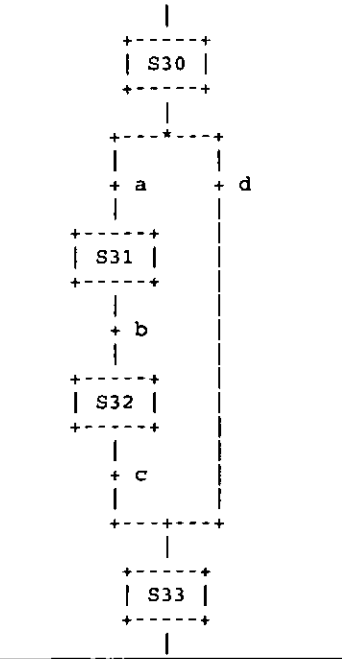
1. O campo "a" pode omitirse cando o tipo da acción é "N".
2. O campo "c" pode omitirse cando non se utilicen indicadores.

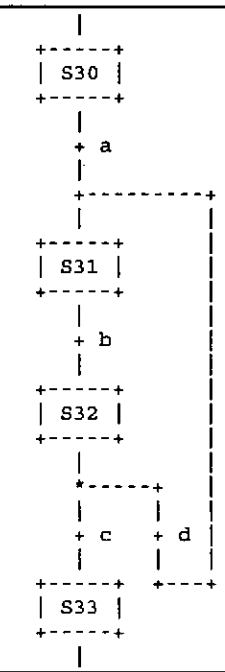
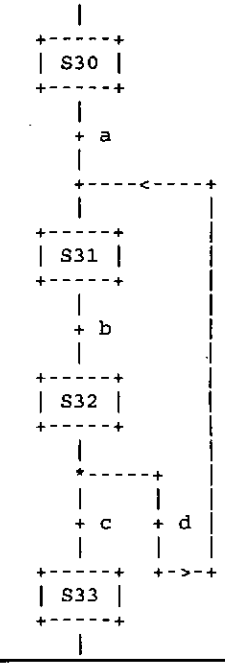
Táboa A-V. Características dos bloques de definición de accións.

Nº	Cualificador	Explicación
1	Ningún	Non almacenado (cualificador nulo)
2	N	Non almacenado
3	R	Reset
4	S	Set
5	L	Limitado
6	D	Demorado
7	P	Pulso
8	SD	Almacenado e demorado
9	DS	Demorado e almacenado
10	SL	Almacenado e limitado

Táboa A-VI. Tipos de accións.

No.	Exemplo	Regra
1		*Secuencia simple*: a alternancia etapa-transición repítese en series. *Exemplo*: a evolución dende S3 a S4 terá lugar só si S3 está activo e a condición de transición *c* é certa.
2a		*Diverxencia nunha selección de secuencia*: a selección entre varias secuencias represéntase mediante tantos símbolos de transición baixo a liña horizontal como posíbeis evolucións haxa. O asterisco indica que a prioridade de avaliación das condicións de transición é de esquerda a dereita. *Exemplo*: a evolución dende S5 a S6 terá lugar só si S5 está activo e a condición de transición *e* é certa, ou dende S5 a S8 se S5 está activo, a condición de transición *f* é certa e a condición *e* é falsa.
2b		*Diverxencia nunha selección de secuencia*: o asterisco seguido de pólas numeradas, indica explicitamente a prioridade de avaliación das condicións de transición. A póla con prioridade mais alta será a que teña o menor número. *Exemplo*: a evolución dende S5 a S8 terá lugar só se S5 está activo e a condición de transición *f* é certa, ou dende S5 a S6 se S5 está activo, a condición de transición *e* é certa e a condición *f* é falsa.
2c		*Diverxencia nunha selección de secuencia*: a conexión das pólas indica que o usuario ten que garantir explicitamente que as condicións de transición son mutuamente exclusivas, tal e como o define o IEC 60848. *Exemplo*: a evolución dende S5 a S6 terá lugar só si S5 está activo e a condición de transición *e* é certa, ou dende S5 a S8 se S5 está activo, a condición de transición *f* é certa e a condición *e* é falsa.

3		*Converxencia nunha selección de secuencia*: o final dunha selección entre varias secuencias represéntase mediante tantos símbolos de transición sobre a liña horizontal como secuencias rematen na converxencia. *Exemplo*: a evolución dende S7 a S10 terá lugar só si S7 está activo e a condición de transición *h* é certa, ou dende S9 a S10 se S9 está activo e a condición *j* é certa.
4		*Diverxencia de secuencias simultáneas*: represéntase mediante un único símbolo de transición sobre a dobre liña horizontal que indica a diverxencia. *Exemplo*: a evolución dende S11 a S12, S14,... terá lugar só si S11 está activo e a condición de transición *b* é certa. Despois da activación simultánea de S12, S14,... cada secuencia evoluciona independentemente.
		*Converxencia de secuencias simultáneas*: represéntase mediante un único símbolo de transición baixo a dobre liña horizontal que indica a converxencia. *Exemplo*: a evolución dende S13,S15... a S16 terá lugar só si todas as etapas que están por riba da dobre liña horizontal e conectadas a ela están activas e a condición de transición *d* é certa.
5a 5b 5c		*Salto de secuencia cara adiante*: este salto é un caso especial da selección de secuencia, no que unha ou mais das pólas non conteñen etapas. As mesmas características aplicábeis á selección de secuencia (2a, 2b e 2c) son tamén aplicábeis ao salto. *Exemplo*: a evolución dende S30 a S33 terá lugar só si a condición de transición *a* é falsa e a condición *d* certa. Desta maneira sáltase a secuencia (S31, S32).

6a 6b 6c		*Salto de secuencia cara atrás*: este salto é un caso especial da selección de secuencia, no que unha ou mais das pólas volven a unha etapa previa. As mesmas características aplicábeis á selección de secuencia (2a, 2b e 2c) son tamén aplicábeis ao salto. *Exemplo*: a evolución dende S32 a S31 terá lugar só si a condición de transición *c* é falsa e a condición *d* certa. Desta maneira a secuencia (S31, S32) repítese.
7		*Arcos orientados*: cando sexa preciso clarificar o sentido dunha conexión entre dous elementos do SFC, pode incluírse na liña que os una unha pequena cabeza de frecha (>) que o indique explicitamente.

Táboa A-VII. Estructuras de control.

Anexo B. Funcionalidades básicas da libraría

Neste anexo descríbense de forma breve as funcionalidades básicas relacionadas co almacenamento e representación de variábeis e tipos de datos implementadas como parte da libraría que da soporte ás diferentes compoñentes da ferramenta desenvolvida nesta tese de doutoramento. Outras funcionalidades que non se incluíron neste anexo e que tamén forman parte da libraría son, por exemplo, os mecanismos de conta de referencias a obxectos ou a implementación da 'persistencia'.

B.1. Contedores

A libraría inclúe contedores con diferentes semánticas no referente á 'persistencia', copia e asignación de elementos, para o manexo de coleccións de obxectos de distintos tipos con identificadores alfanuméricos únicos. A implementación destes contedores está baseada nas coleccións definidas como parte da libraría STL [11][123]. Na Figura B.1 pode verse o diagrama das clases utilizadas para representar os distintos tipos de contedores implementados, e na Táboa B-I resúmense as propiedades de cada un.

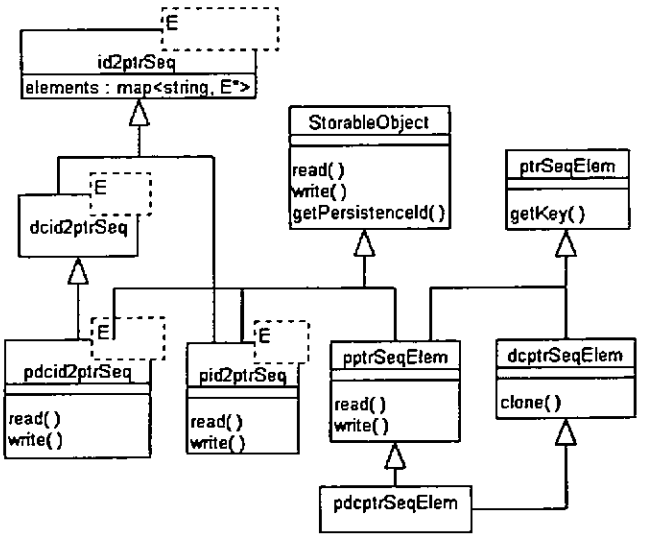


Figura B.1. Diagrama de clases dos contedores implementados na libraría.

Colección	Interface elementos	Persistencia	Copia, asignación
id2ptrSeq	ptrSeqElem	Non	Por referencia
dcid2ptrSeq	dcptrSeqElem	Non	Por valor
pid2ptrSeq	pptrSeqElem	Si	Por referencia
pdcid2ptrSeq	pdcptrSeqElem	Si	Por valor

Táboa B-I. Propiedades dos contedores implementados na libraría.

A clase *id2ptrSeq* é unha clase parametrizada que define e implementa a interface básica para o manexo dunha colección de obxectos de tipo *E* (e derivados) identificados mediante unha chave alfanumérica única. Esta clase non implementa a persistencia dos elementos que almacena, e as operacións de copia e asignación de elementos son unicamente por referencia ("shallow copy"). A interface pública desta clase é a seguinte:

```
3329. template <class E>
3330.     class id2ptrSeq
3331. {
3332. public:
3333.  // declaración de referencias e apuntadores aos elementos da colección
3334.  typedef E* pointer;
3335.  typedef const E* const_pointer;
3336.  typedef E& reference;
3337.  typedef const E& const_reference;
3338.  // constructores/destructor
3339.  id2ptrSeq();                          // constructor
3340.  id2ptrSeq(const id2ptrSeq& seq); // constructor de copia
3341.  virtual ~id2ptrSeq();                 // destructor
3342.  // consulta
3343.  bool empty() const;                   // ¿colección baleira?
3344.  unsigned size() const;                // número de elementos
3345.  // asignación/inserción elementos
3346.  virtual void assign(const_iterator first, const_iterator last);
3347.  iterator insert(pointer element);
3348.  // eliminación elementos
3349.  unsigned erase(const string& key);
3350.  void erase(iterator element);
3351.  void erase(iterator first, iterator last);
3352.  // acceso aos extremos da colección
3353.  iterator begin();
3354.  iterator end();
3355.  reverse_iterator rbegin();
3356.  reverse_iterator rend();
3357.  const_iterator begin() const;
3358.  const_iterator end() const;
3359.  const_reverse_iterator rbegin() const;
3360.  const_reverse_iterator rend() const;
3361.  // busca de elementos
3362.  iterator find(const string& key);
3363.  const_iterator find(const string& key) const;
3364.  // outras operacións
3365.  void clear();                  // baleirar colección (sen destruir elementos)
3366.  void destroy();                // baleirar colección (destruindo elementos)
3367.  void swap(id2ptrSeq& seq);     // intercambiar elementos entre coleccións
3368.  // operadores
3369.  id2ptrSeq& operator= (const id2ptrSeq& seq);          // asignación
3370.  bool operator== (const id2ptrSeq& seq) const;         // igualdade
3371.  reference operator[](const string& key);              // referencia
3372.  const_reference operator[](const string& key) const; // referencia constante
3373. };
```

Internamente esta clase capsula un mapa STL con chaves tipo *string* e elementos de tipo *E** —atributo *elements* (Figura B.1)—, e os métodos da súa interface non son mais que adaptadores dos métodos equivalentes no mapa. A clase tamén define iteradores bidireccionais (non incluídos aquí) que capsulan aos do mapa e permiten utilizar a colección como se se tratase dun contedor de elementos tipo *E*, ocultando os detalles do manexo de apuntadores aos elementos. Na implementación dos métodos da clase *id2ptrSeq* presuponse que os elementos almacenados na colección implementan a interface seguinte:

```
3374. struct ptrSeqElem
3375. {
3376.   virtual const string& getKey(void) const = 0;
3377. };
```

O método *getKey* devolve o identificador alfanumérico único do elemento. O seguinte código mostra un exemplo de utilización da clase *id2ptrSeq*:

```
3378. // Declaración da clase dos elementos
3379. class Element : public ptrSeqElem {
3380.   string key;
3381. public:
3382.   Element(const string& k) : key(k){}
3383.   const string& getKey(void) const {return key;}
3384. }
3385.
3386. // Instanciación da colección e o iterador
3387. typedef id2ptrSeq<Element> ElementSeq;
3388. typedef id2ptrSeq<Element>::const_iterator const_element_iterator;
3389.
3390. // Utilización da colección
3391. main()
3392. {
3393.   // declaración da colección
3394.   ElementSeq elements;
3395.   // declaración dos elementos
3396.   Element e1("first");
3397.   Element e2("second");
3398.   // inserimento dos elementos na colección
3399.   elements.insert(&e1);
3400.   elements.insert(&e2);
3401.   // percorrido e acceso ao identificador dos elementos
3402.   for (const_element_iterator iter = elements.begin();
3403.         iter != elements.end();
3404.         ++iter)
3405.     cout << iter->getKey();
3406. }
```

Os demais contedores implementados son definidos mediante "templates" derivados de *id2ptrSeq*. O contedor *pid2ptrSeq* (Figura B.1) engade o soporte á persistencia dos elementos almacenados na colección. A declaración pública da súa interface é a seguinte:

```
3407. template <class E>
3408.    class pid2ptrSeq : public id2ptrSeq<E>, public StorableObject
3409. {
3410.   GESCA_DECLARETEMPLATE_PERSISTENT
3411. public:
3412.   virtual bool write(Storage& st) const;
3413.   virtual bool read(Storage& st);
3414. };
3415.
3416. GESCA_IMPLEMENTTEMPLATE1_PERSISTENT(pid2ptrSeq, E)
```

Toda clase que utilice o mecanismo de 'persistencia' implementado pola libraría ten que ser derivada da interface *StorableObject* (Figura B.1), e implementar os métodos *read* e *write* que son os que xestionan a 'serialización' das instancias da clase. Estes métodos reciben como parámetro unha instancia da clase *Storage*, que é a que manexa os detalles da implementación da 'persistencia'. As macros GESCA_DECLARETEMPLATE_PERSISTENT e GESCA_IMPLEMENTTEMPLATE1_PERSISTENT serven para incluír nas coleccións 'serializábeis' a declaración e implementación de métodos auxiliares utilizados polo mecanismo de 'persistencia'. A implementación da clase *pid2ptrSeq* presupón que os elementos que almacena implementan a interface seguinte, que engade as operacións de 'serialización' do elemento ás definidas pola interface *ptrSeqElem*:

```
3417. struct pptrSeqElem : public virtual ptrSeqElem, public virtual StorableObject
3418. {
3419.   virtual bool write(Storage& st) const = 0;
3420.   virtual bool read(Storage& st) = 0;
3421. };
```

Os outros dous contedores implementados na libraría: *dcid2ptrSeq* e *pdcid2ptrSeq* (Figura B.1), son especializacións dos anteriores que implementan as operacións de copia e asignación cunha semántica por valor ("deep copy"). A implementación dambos contedores presupón que os elementos que almacenan implementan a interface seguinte, que engade a operación de 'clonación' do elemento ás definidas pola interface *ptrSeqElem*:

```
3422. struct dcptrSeqElem : public virtual ptrSeqElem
3423. {
3424.   virtual dcptrSeqElem* clone(void) const = 0;
3425. };
```

O método *clone* devolve unha copia do elemento. O código seguinte mostra exemplos da declaración de elementos e coleccións utilizando os distintos contedores implementados na libraría:

```
3426. // Colección sen 'persistencia' e con copia por referencia
3427. class Element : public ptrSeqElem
3428. {
3429.   string key;
3430. public:
3431.   Element(const string& k) : key(k){}
3432.   const string& getKey(void) const {return key;}
3433. }
3434.
3435. typedef id2ptrSeq<Element> ElementSeq;
3436. typedef id2ptrSeq<Element>::iterator element_iterator;
3437.
3438. // Colección con 'persistencia' e con copia por referencia
3439. class Element : public pptrSeqElem
3440. {
3441.   string key;
3442. public:
3443.   Element(const string& k) : key(k){}
3444.   const string& getKey(void) const {return key;}
3445.   bool write(Storage& st) const { st << key; return true;}
3446.   bool read(Storage& st) { st >> key; return true;}
3447. }
3448.
3449. typedef pid2ptrSeq<Element> ElementSeq;
3450. typedef pid2ptrSeq<Element>::iterator element_iterator;
```

```
3451. // Colección sen 'persistencia' e con copia por valor
3452. class Element : public dcptrSeqElem
3453. {
3454.   string key;
3455. public:
3456.   Element(const string& k) : key(k){}
3457.   Element(const Element& e) : key(e.key){}
3458.   const string& getKey(void) const {return key;}
3459.   dcptrSeqElem* clone(void) const {return new Element(this);}
3460. }
3461.
3462. typedef dcid2ptrSeq<Element> ElementSeq;
3463. typedef dcid2ptrSeq<Element>::iterator element_iterator;
3464.
3465. // Colección con 'persistencia' e con copia por valor
3466. class Element : public pdcptrSeqElem
3467. {
3468.   string key;
3469. public:
3470.   Element(const string& k) : key(k){}
3471.   Element(const Element& e) : key(e.key){}
3472.   const string& getKey(void) const {return key;}
3473.   dcptrSeqElem* clone(void) const {return new Element(this);}
3474.   bool write(Storage& st) const { st << key; return true;}
3475.   bool read(Storage& st) { st >> key; return true;}
3476. }
3477.
3478. typedef pdcid2ptrSeq<Element> ElementSeq;
3479. typedef pdcid2ptrSeq<Element>::iterator element_iterator;
```

B.2. Variábeis e declaracións

A libraría inclúe diferentes clases para manexar os distintos tipos de variábeis e declaracións utilizados na ferramenta implementada. Como pode verse na Figura B.2, todas as clases definidas son derivadas da interface *pdcptrSeqElem*, polo que poden ser utilizadas conxuntamente cos contedores explicados no apartado anterior. A descrición detallada das clases definidas é a seguinte:

ElementAccess

Enumeración que define o tipo de acceso permitido ao valor dunha variábel. Os valores escalares definidos son:

- *read*, o valor pode ser consultado, mais non modificado.
- *write*, o valor pode ser modificado, mais non consultado.
- *readwrite*, o valor pode ser consultado e modificado.

ElementScope

Enumeración que define o alcance dunha variábel en relación ao modelo no que é utilizada. Os valores escalares definidos son:

- *local*, a variábel é definida no contexto do modelo e só pode ser utilizada nese contexto.
- *shared*, a variábel é definida no contexto do modelo e pode ser utilizada fora dese contexto.
- *external*, a variábel é utilizada no modelo mais é definida fora do seu contexto.

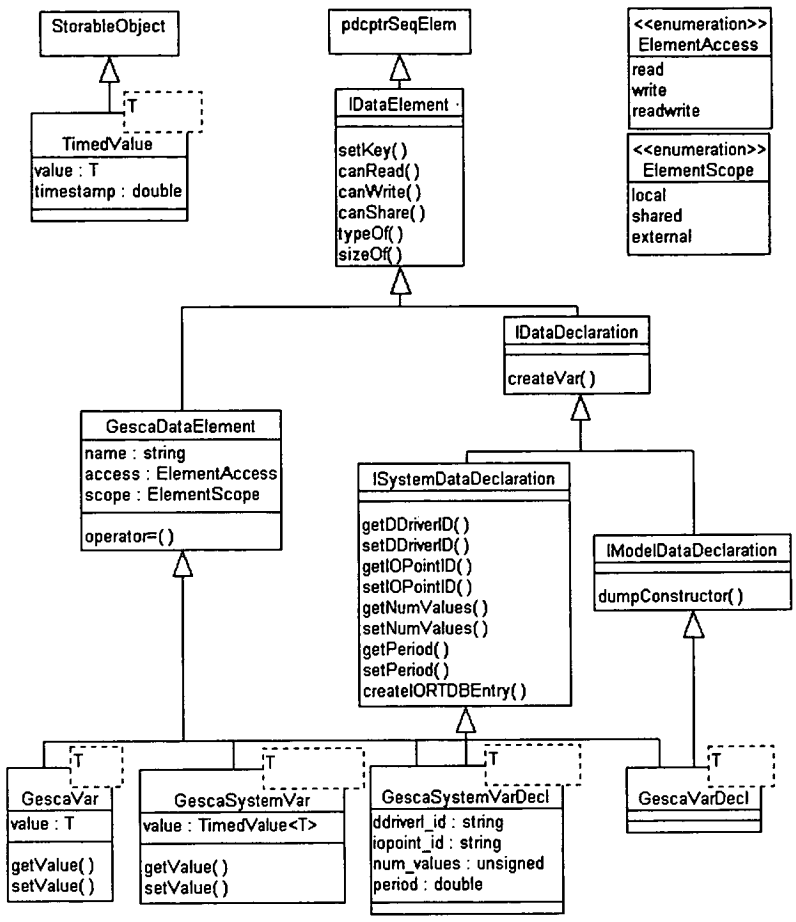


Figura B.2. Diagrama de clases das variábeis e declaracións implementadas na libraría.

IDataElement

Clase abstracta que define a interface común a todas as clases da libraría utilizadas para representar as variábeis e as declaracións de variábeis. A súa declaración é a seguinte:

```
3480. struct IDataElement : public pdcptrSeqElem
3481. {
3482.  // modificación do identificador
3483.  virtual void setKey(const string& name) = 0;
3484.  // consulta do tipo de acceso e alcance
3485.  virtual bool canReadValue() const = 0;
3486.  virtual bool canWriteValue() const = 0;
3487.  virtual bool canShareValue() const = 0;
3488.  // consulta do tipo de datos da variábel
3489.  virtual const type_info& typeOf() const = 0;
3490.  virtual size_t sizeOf() const = 0;
3491. };
3492.
3493. // Apuntadores e referencias a elementos de datos
3494. typedef IDataElement data_element;
3495. typedef IDataElement* data_pointer;
3496. typedef IDataElement& data_reference;
3497. typedef const IDataElement* const_data_pointer;
3498. typedef const IDataElement& const_data_reference;
```

IDataDeclaration

Clase abstracta que define a interface común a todas as clases utilizadas para representar declaracións de variábeis. A súa declaración é a seguinte:

```
3499. struct IDataDeclaration : virtual public IDataElement
3500. {
3501.  // método constructor (variábel do tipo declarado)
3502.  virtual data_pointer CreateVar() const = 0;
3503. };
```

Esta interface declara o método *CreateVar*, que é un método constructor ("factory method", [67]) utilizado para crear unha instancia do tipo de variábel que corresponde á declaración.

IModelDataDeclaration

Clase abstracta que define a interface das clases utilizadas para representar as declaracións de variábeis internas aos modelos. A súa declaración é a seguinte:

```
3504. struct IModelDataDeclaration : virtual public IDataDeclaration
3505. {
3506.  virtual void dumpConstructor(ostream& out) const = 0;
3507. };
```

Esta interface declara o método *dumpConstructor*, que devolve unha representación textual do código C++ que crea unha instancia do tipo de variábel que corresponde á declaración. Este método é utilizado polo compilador Grafcet (§6.5.1.9) para xerar o código das declaracións de variábeis do modelo (liñas 1498-1503).

ISystemDataDeclaration

Clase abstracta que define a interface das clases utilizadas para representar as declaracións de variábeis do proceso utilizadas nos modelos. A súa declaración é a seguinte:

```
3508. struct ISystemDataDeclaration : virtual public IDataDeclaration
3509. {
3510.  // "driver" no que a variábel está asignada
3511.  virtual void setDDriverID(const string& dd) = 0;
3512.  virtual const string& getDDriverID(void) const = 0;
3513.  // punto de E/S no que a variábel está asignada
3514.  virtual void setIOPointID(const string& io_point) = 0;
3515.  virtual const string& getIOPointID(void) const = 0;
3516.  // número de valores históricos almacenados na base de datos de E/S
3517.  virtual void setNumValues(unsigned num) = 0;
3518.  virtual unsigned getNumValues(void) const = 0;
3519.  // frecuencia de monitorización
3520.  virtual void setPeriod(double period) = 0;
3521.  virtual double getPeriod(void) const = 0;
3522.  // método constructor (entrada na base de datos de E/S)
3523.  virtual IIORTDBEntry* CreateIORTDBEntry() const = 0;
3524. };
```

Esta interface declara métodos para acceder aos valores que identifican o "driver" e o punto de E/S no que a variábel vai estar asignada (§7.2.1.4.1), a frecuencia de monitorización do valor da variábel (§7.2.1.4.2), e o número de valores históricos a almacenar na base de datos de E/S da máquina virtual (§7.3.1.1). Ademais declárase un método constructor — *CreateIORTDBEntry*— utilizado para crear unha entrada na base de datos de E/S que se corresponda cos datos almacenados na declaración.

GescaDataElement

Clase abstracta que implementa parte da funcionalidade común a todas as clases utilizadas para representar variábeis e declaracións de variábeis. En concreto esta clase declara atributos (Figura B.2) para almacenar o identificador —atributo *name*—, o tipo de acceso —atributo *access*— e o alcance —atributo *scope*— da variábel ou declaración, e proporciona un operador de asignación e implementacións por defecto para os métodos *getKey*, *read* e *write* herdados da interface *pdcptrSeqElem*; e para os métodos *setKey*, *canRead*, *canWrite* e *canShare* herdados da interface *IDataElement*.

GescaVarDecl

Clase parametrizada utilizada para representar as declaracións das variábeis internas dos modelos. As declaracións dos diferentes tipos de variábeis obtéñense por instanciación (Figura 5.12), indicando como argumento o tipo de dato a almacenar na variábel declarada.

GescaVar

Clase parametrizada utilizada para representar as variábeis internas dos modelos. Esta clase declara un atributo (Figura B.2) no que se almacena o valor da variábel —atributo *value*— e métodos para consultar —método *getValue*— e modificar —método *setValue*— o seu valor. Os diferentes tipos de variábeis obtéñense por instanciación, indicando como argumento o tipo de dato a almacenar.

GescaSystemVarDecl

Clase parametrizada utilizada para representar as declaracións das variábeis do proceso utilizadas nos modelos. Esta clase declara atributos (Figura B.2) para almacenar as propiedades da variábel: o "driver" —atributo *ddriver_id*— e o punto de E/S —atributo *iopoint_id*— no que a variábel vai estar asignada (§7.2.1.4.1), a frecuencia de monitorización —atributo *period*— do valor da variábel (§7.2.1.4.2), e o número de valores históricos —atributo *num_values*— a almacenar na base de datos de E/S da máquina virtual (§7.3.1.1). As declaracións dos diferentes tipos de variábeis obtéñense por instanciación, indicando como argumento o tipo de dato a almacenar na variábel declarada.

GescaSystemVar

Clase parametrizada utilizada para representar as variábeis do proceso utilizadas nos modelos. Esta clase declara un atributo (Figura B.2) no que se almacena o valor da variábel —atributo *value*— e métodos para consultar —método *getValue*— e modificar —método *setValue*— o seu valor. Nótese que, a diferencia da clase *GescaVar*, o valor almacenado é unha instancia da clase parametrizada *TimedValue* que almacena o valor xunto coa data e hora na que foi obtido. Os diferentes tipos de variábeis obtéñense por instanciación, indicando como argumento o tipo de dato a almacenar.

TimedValue

Clase parametrizada utilizada para representar os valores das variábeis do proceso utilizadas nos modelos. Esta clase declara atributos (Figura B.2) para almacenar tanto o valor da variábel —atributo *value*— como a data e a hora na que foi obtido —atributo *time_stamp*—. Os diferentes tipos de valores obtéñense por instanciación, indicando como argumento o tipo de dato a almacenar.

Anexo C. Compilación da DLL xerada no compilador Grafcet

Exemplo de arquivo *.mak* utilizado para compilar e enlazar co Visual C++ 6.0 o código dunha DLL (§6.5.1.9) xerado polo compilador Grafcet.

```
3525. #
3526. # arquivo make para compilar unha DLL que contén un modelo Grafcet
3527. #
3528.
3529. !IF "$(CFG)" == ""
3530. CFG=Debug
3531. !MESSAGE No configuration specified. Defaulting to Debug.
3532. !ENDIF
3533.
3534. !IF "$(CFG)" != "Release" && "$(CFG)" != "Debug"
3535. !MESSAGE
3536. !MESSAGE Possible choices for configuration are:
3537. !MESSAGE
3538. !MESSAGE "Release" (based on "Win32 (x86) Dynamic-Link Library")
3539. !MESSAGE "Debug" (based on "Win32 (x86) Dynamic-Link Library")
3540. !MESSAGE
3541. !ERROR Invalid configuration "$(CFG)" specified.
3542. !ENDIF
3543.
3544. !IF "$(DLLNAME)" == ""
3545. !ERROR No model name "$(DLLNAME)" has been specified.
3546. !ENDIF
3547.
3548. !IF "$(SOURCEPATH)" == ""
3549. !ERROR No source path "$(SOURCEPATH)" has been specified.
3550. !ENDIF
3551.
3552. !IF "$(SFCPPPATH)" == ""
3553. !ERROR No SFCPP path "$(SFCPPPATH)" has been specified.
3554. !ENDIF
3555.
3556. TMPDIR=.\Compiled
3557. CPP=cl.exe
3558. MTL=midl.exe
3559. RSC=rc.exe
3560. BSC32=bscmake.exe
3561. LINK32=link.exe
```

```
3562. BSC32_FLAGS=/nologo /o"$(TMPDIR)\\$(DLLNAME).bsc"
3563. BSC32_SBRS= \
3564.
3565. !IF  "$(CFG)" == "Release"
3566. CPP_PROJ=/nologo /MD /W3 /Gi /GR /GX /O2 /D "WIN32" /D "NDEBUG" /D "_WINDOWS" /D
3567.         "_MBCS" /D "_USRDLL" /I"$(SOURCEPATH)" /I"$(SFCPPPATH)\\include"
3568.         /I"$(SFCPPPATH)\\include\\stlport" /I"$(SFCPPPATH)\\include\\commoncpp"
3569.         /Fp"$(TMPDIR)\\$(DLLNAME).pch" /YX /Fo"$(TMPDIR)\\$(DLLNAME).obj"
3570.         /Fd"$(TMPDIR)\\" /FD /c
3571. MTL_PROJ=/nologo /D "NDEBUG" /mktyplib203 /win32
3572. LINK32_FLAGS=/LIBPATH:"$(SFCPPPATH)\\Lib" stlport_vc6.lib ccppw32dll.lib
3573.         gescalib.lib gescadata.lib gescapersistence.lib gescaruntimelib.lib
3574.         gescaruntimeinfo.lib gescavmlib.lib gescaiortdb.lib gescasfcplayer.lib
3575.         /nologo /dll /incremental:no /pdb:"$(TMPDIR)\\$(DLLNAME).pdb"
3576.         /machine:I386 /nodefaultlib:"nafxcw.lib"
3577.         /out:"$(TMPDIR)\\$(DLLNAME).dll" /implib:"$(TMPDIR)\\$(DLLNAME).lib"
3578.         /pdbtype:sept
3579. !ELSEIF  "$(CFG)" == "Debug"
3580. CPP_PROJ=/nologo /MDd /W3 /Gm /Gi /GR /GX /ZI /Od /D "WIN32" /D "_DEBUG" /D
3581.         "_WINDOWS" /D "_MBCS" /D "_USRDLL" /D "_MT" /D "_DLL" /D "_WINDLL" /D
3582.         "_AFXDLL" /I"$(SOURCEPATH)" /I"$(SFCPPPATH)\\include"
3583.         /I"$(SFCPPPATH)\\include\\stlport" /I"$(SFCPPPATH)\\include\\commoncpp"
3584.         /Fo"$(TMPDIR)\\$(DLLNAME).obj" /Fd"$(TMPDIR)\\" /FD /GZ /Zm150 /c
3585. MTL_PROJ=/nologo /D "_DEBUG" /mktyplib203 /win32
3586. LINK32_FLAGS=/LIBPATH:"$(SFCPPPATH)\\Lib" stlport_vc6.lib ccppw32dll.lib
3587.         gescalib.lib gescadata.lib gescapersistence.lib gescaruntimelib.lib
3588.         gescaruntimeinfo.lib gescavmlib.lib gescaiortdb.lib gescasfcplayer.lib
3589.         /nologo /dll /incremental:yes /pdb:"$(TMPDIR)\\$(DLLNAME).pdb" /debug
3590.         /machine:I386 /nodefaultlib:"nafxcwd.lib"
3591.         /out:"$(TMPDIR)\\$(DLLNAME).dll" /implib:"$(TMPDIR)\\$(DLLNAME).lib"
3592.         /pdbtype:sept
3593. !ENDIF
3594.
3595. .c{$(TMPDIR)}.obj::
3596.    $(CPP) @<<
3597.    $(CPP_PROJ) $<
3598. <<
3599.
3600. .cpp{$(TMPDIR)}.obj::
3601.    $(CPP) @<<
3602.    $(CPP_PROJ) $<
3603. <<
3604.
3605. .cxx{$(TMPDIR)}.obj::
3606.    $(CPP) @<<
3607.    $(CPP_PROJ) $<
3608. <<
3609.
3610. .c{$(TMPDIR)}.sbr::
3611.    $(CPP) @<<
3612.    $(CPP_PROJ) $<
3613. <<
3614.
3615. .cpp{$(TMPDIR)}.sbr::
3616.    $(CPP) @<<
3617.    $(CPP_PROJ) $<
3618. <<
3619.
3620. .cxx{$(TMPDIR)}.sbr::
3621.    $(CPP) @<<
3622.    $(CPP_PROJ) $<
3623. <<
3624.
3625. LINK32_OBJS="$(TMPDIR)\\$(DLLNAME).obj"
3626. SOURCE="$(TMPDIR)\\$(DLLNAME)DLL.cpp"
```

```
3627. ALL : "$(TMPDIR)" "$(TMPDIR)\\$(DLLNAME).dll"
3628.
3629. "$(TMPDIR)" :
3630.   cd "$(SOURCEPATH)"
3631.
3632. "$(TMPDIR)\\$(DLLNAME).dll" : $(LINK32_OBJS)
3633.     $(LINK32) @$(USER_LIBS) @<<
3634.   $(LINK32_FLAGS) $(LINK32_OBJS)
3635. <<
3636.
3637. "$(TMPDIR)\\$(DLLNAME).obj" :
3638.   $(CPP) $(CPP_PROJ) $(SOURCE)
3639.
3640. CLEAN :
3641.   -@erase "$(TMPDIR)\\$(DLLNAME).obj"
3642.   -@erase "$(TMPDIR)\\vc60.idb"
3643.   -@if exits "$(TMPDIR)\\vc60.pdb" erase "$(TMPDIR)\\vc60.pdb"
3644.   -@erase "$(TMPDIR)\\$(DLLNAME).dll"
3645.   -@erase "$(TMPDIR)\\$(DLLNAME).exp"
3646.   -@erase "$(TMPDIR)\\$(DLLNAME).lib"
3647.   -@if exits "$(TMPDIR)\\$(DLLNAME).pdb" erase "$(TMPDIR)\\$(DLLNAME).pdb"
```

Anexo D. Gramática ANTLR

Neste anexo móstrase a gramática ANTLR [135] utilizada para xerar automaticamente un analizador sintáctico para as expresións C++ estendidas cos operadores de evento e temporización, tal e como explica en (§6.5.1.5.3). As regras da gramática conteñen ademais as anotacións e accións utilizadas para construír a AST da expresión ao tempo que se realiza a súa análise sintáctica.

```
3648. /*
3649.  *  Gramática C++ para ANTLR e SORCERER
3650.  *  Expresións C++ sen:
3651.  *    . especializacións de "templates"
3652.  *    . "casts"
3653.  *    . conversións de tipo explícitas
3654.  *    . operadores sizeof e typeid
3655.  *    . funcións membro operator
3656.  *    . funcións membro de conversión
3657.  *    . operadores de reserva e liberación de memoria
3658.  *    . modificadores de Microsoft nos nomes cualificados
3659.  *  Engadíronse:
3660.  *    . operadores de evento: UP_EDGE e DOWN_EDGE
3661.  */
3662. #header
3663. <<
3664.   #include <deque>
3665.   #include <string>
3666.   #include <set>
3667.   #include <sstream>
3668.   using namespace std;
3669.   #include "tokens.h"
3670.   #include "AToken.h"
3671.   #include "ATokenBuffer.h"
3672.   typedef ANTLRCommonToken ANTLRToken;
3673.   #include "ASTBase.h"
3674.   #include "ATokPtr.h"
3675.   #include "AST.h"
3676.   typedef AST SORAST;
3677. >>
3678.
3679. // "TOKENS" SIMPLES
3680.
3681. // ANSI C++ "tokens"
3682. #token TOK_AMPERSAND            "&"
3683. #token TOK_AND                  "&&"
3684. #token TOK_ASSIGNEQUAL          "="
3685. #token TOK_BITWISEANDEQUAL      "&="
3686. #token TOK_BITWISEOR            "\|"
```

```
3687. #token TOK_BITWISEOREQUAL        "\|="
3688. #token TOK_BITWISEXOR            "^"
3689. #token TOK_BITWISEXOREQUAL       "^="
3690. #token TOK_COMMA                 ","
3691. #token TOK_COLON                 ":"
3692. #token TOK_DIVIDE                "/"
3693. #token TOK_DIVIDEEQUAL           "/="
3694. #token TOK_DOT                   "."
3695. #token TOK_DOTMBR                ".\*"
3696. #token TOK_ELLIPSIS              "..."
3697. #token TOK_EQUAL                 "=="
3698. #token TOK_GREATERTHAN           ">"
3699. #token TOK_GREATERTHANOREQUALTO  ">="
3700. #token TOK_LCURLY                "\{"
3701. #token TOK_LESSTHAN              "<"
3702. #token TOK_LESSTHANOREQUALTO     "<="
3703. #token TOK_LPAREN                "\("
3704. #token TOK_LSQUARE               "\["
3705. #token TOK_MINUS                 "\-"
3706. #token TOK_MINUSEQUAL            "\-="
3707. #token TOK_MINUSMINUS            "\-\-"
3708. #token TOK_MOD                   "%"
3709. #token TOK_MODEQUAL              "%="
3710. #token TOK_NOT                   "!"
3711. #token TOK_NOTEQUAL              "!="
3712. #token TOK_OR                    "\|\|"
3713. #token TOK_PLUS                  "\+"
3714. #token TOK_PLUSEQUAL             "\+="
3715. #token TOK_PLUSPLUS              "\+\+"
3716. #token TOK_POINTERTO             "\->"
3717. #token TOK_POINTERTOMBR          "\->\*"
3718. #token TOK_QUESTIONMARK          "?"
3719. #token TOK_RCURLY                "\}"
3720. #token TOK_RPAREN                "\)"
3721. #token TOK_RSQUARE               "\]"
3722. #token TOK_SCOPE                 "::"
3723. #token TOK_SEMICOLON             ";"
3724. #token TOK_SHIFTLEFT             "\<\<"
3725. #token TOK_SHIFTLEFTEQUAL        "\<\<="
3726. #token TOK_SHIFTRIGHT            "\>\>"
3727. #token TOK_SHIFTRIGHTEQUAL       "\>\>="
3728. #token TOK_STAR                  "\*"
3729. #token TOK_TILDE                 "\~"
3730. #token TOK_TIMESEQUAL            "\*="
3731.
3732. // C++ extensions
3733. #token TOK_UPEDGE                "\0x18"
3734. #token TOK_DWEDGE                "\0x19"
3735.
3736. // ANSI C++ Keywords
3737. #token TOK_BOOLFALSE             "false"
3738. #token TOK_BOOLTRUE              "true"
3739. #token TOK_THIS                  "this"
3740.
3741. // "TOKENS" COMPLEXOS
3742.
3743. #token TOK_ID                    "[a-zA-Z_][a-zA-Z0-9_]*"
3744. #token TOK_OCTALINT              "0[0-7]*{[uUlL]}"
3745. #token TOK_DECIMALINT            "[1-9][0-9]*{[uUlL]}"
3746. #token TOK_HEXADECIMALINT        "(0x|0X)[0-9a-fA-F]+{[uUlL]}"
3747. #token TOK_FLOATONE              "([0-9]+ . [0-9]* | [0-9]* . [0-9]+)
3748.                                   {[eE]{[\-\+]}[0-9]+} {[fFlL]}"
3749. #token TOK_FLOATTWO              "[0-9]+ [eE] {[\-\+]} [0-9]+ {[fFlL]}"
```

```
3750. #token "\""                       << mode(STRINGS); more (); >>
3751. #token "'"                        << mode(CHARACTERS); more (); >>
3752. #token "\\\n"                     << newline(); skip(); >>
3753. #token "[\t\ ]+"                  << skip(); >>
3754. #token "\12"                      << skip(); >>
3755. #token TOK_NEWLINE "\n"           << newline(); skip(); >>
3756. #token TOK_EOF "@"
3757.
3758. // CLASES LÉXICAS AUXILIARES
3759.
3760. // Arranxos de caracteres
3761. #lexclass STRINGS
3762. #token TOK_STRING "\""             << mode (START); >>
3763. #token " \\ ( [ntvbrfa\\?'\"] | [0-9]+ | (x|X)[0-9a-fA-F]+ ) "  << more(); >>
3764. #token "\\ [\n\r]"                 << newline(); more(); >>
3765. #token "~[\\ \" \n \r]+"          << more(); >>
3766.
3767. // Constantes de caracter
3768. #lexclass CHARACTERS
3769. #token TOK_CHARACTER "'"           << mode (START); >>
3770. #token " \\ ( [ntvbrfa\\?'\"] | [0-9]+ | (x|X)[0-9a-fA-F]+ ) "  << more(); >>
3771. #token "~[\\ ' \n \r]+"           << more(); >>
3772.
3773. // Clase inicial
3774. #lexclass START
3775.
3776. // "Tokens" de asignación
3777. #tokclass ASSIGN_TOKCLASS
3778. {
3779.    TOK_ASSIGNEQUAL TOK_TIMESEQUAL TOK_DIVIDEEQUAL
3780.    TOK_MODEQUAL TOK_PLUSEQUAL TOK_BITWISEXOREQUAL
3781.    TOK_MINUSEQUAL TOK_SHIFTLEFTEQUAL
3782.    TOK_SHIFTRIGHTEQUAL TOK_BITWISEANDEQUAL TOK_BITWISEOREQUAL
3783. }
3784.
3785. // "Tokens" utilizados nas expresións
3786. #tokclass BOOLEANUNARYOP_TOKCLASS
3787. {
3788.    TOK_NOT
3789. }
3790.
3791. #tokclass BOOLEANEVENTOP_TOKCLASS
3792. {
3793.    TOK_UPEDGE TOK_DWEDGE
3794. }
3795.
3796. #tokclass NUMERICUNARYOP_TOKCLASS
3797. {
3798.    TOK_AMPERSAND TOK_STAR TOK_PLUS TOK_MINUS
3799.    TOK_TILDE TOK_PLUSPLUS TOK_MINUSMINUS
3800. }
3801.
3802. #tokclass UNARYOP_TOKCLASS
3803. {
3804.    TOK_AMPERSAND TOK_STAR TOK_PLUS TOK_MINUS TOK_TILDE TOK_NOT
3805.    TOK_PLUSPLUS TOK_MINUSMINUS TOK_UPEDGE TOK_DWEDGE
3806. }
3807.
3808. #tokclass ADDITIVE_TOKCLASS
3809. {
3810.   TOK_PLUS TOK_MINUS
3811. }
```

```
3812. #tokclass EQUALITY_TOKCLASS
3813. {
3814.    TOK_NOTEQUAL TOK_EQUAL
3815. }
3816.
3817. #tokclass MULTIPLICATIVE_TOKCLASS
3818. {
3819.    TOK_STAR TOK_DIVIDE TOK_MOD
3820. }
3821.
3822. #tokclass PM_TOKCLASS
3823. {
3824.    TOK_DOTMBR TOK_POINTERTOMBR
3825. }
3826.
3827. #tokclass RELATIONAL_TOKCLASS
3828. {
3829.    TOK_LESSTHAN TOK_GREATERTHAN
3830.    TOK_LESSTHANOREQUALTO TOK_GREATERTHANOREQUALTO
3831. }
3832.
3833. #tokclass SHIFT_TOKCLASS
3834. {
3835.    TOK_SHIFTLEFT TOK_SHIFTRIGHT
3836. }
3837.
3838. // "Tokens" literais
3839. #tokclass CONSTANTNUMBER_TOKCLASS
3840. {
3841.    TOK_OCTALINT TOK_DECIMALINT TOK_HEXADECIMALINT
3842.    TOK_FLOATONE TOK_FLOATTWO
3843. }
3844.
3845. #tokclass CONSTANTBOOL_TOKCLASS
3846. {
3847.    TOK_BOOLTRUE TOK_BOOLFALSE
3848. }
3849.
3850. // clases SORCERER
3851. #token TOK_BOOLEANUNARYOP
3852. #token TOK_BOOLEANEVENTOP
3853. #token TOK_NUMERICUNARYOP
3854. #token TOK_POSTOP
3855. #token TOK_NUMERICEXPR
3856. #token TOK_EVENTEXPR
3857.
3858. // DEFINICIÓN DO ANALIZADOR SINTÁCTICO
3859.
3860. class CPPParser {
3861.
3862. // Gramática
3863.
3864. sfcpp_expression :
3865.  { expression } TOK_EOF!
3866.  ;
3867.
3868. expression_list :
3869.  expression
3870.  ;
3871.
3872. expression :
3873.  expr_assignment ( TOK_COMMA^ expr_assignment )*
3874.  ;
```

```
3875. expr_assignment :
3876.   expr_conditional { ASSIGN_TOKCLASS^ expr_assignment }
3877.   ;
3878.
3879. expr_conditional :
3880.   expr_logical_or { TOK_QUESTIONMARK^ expression TOK_COLON expr_conditional }
3881.   ;
3882.
3883. expr_logical_or :
3884.   expr_logical_and ( TOK_OR^ expr_logical_and )*
3885.   ;
3886.
3887. expr_logical_and :
3888.   expr_inclusive_or ( TOK_AND^ expr_inclusive_or )*
3889.   ;
3890.
3891. expr_inclusive_or :
3892.   expr_exclusive_or ( TOK_BITWISEOR^ expr_exclusive_or )*
3893.   ;
3894.
3895. expr_exclusive_or :
3896.   expr_and ( TOK_BITWISEXOR^ expr_and )*
3897.   ;
3898.
3899. expr_and :
3900.   expr_equality ( TOK_AMPERSAND^ expr_equality )*
3901.   ;
3902.
3903. expr_equality :
3904.   expr_relational ( EQUALITY_TOKCLASS^ expr_relational )*
3905.   ;
3906.
3907. expr_relational :
3908.   expr_shift ( RELATIONAL_TOKCLASS^ expr_shift )*
3909.   ;
3910.
3911. expr_shift :
3912.   expr_additive ( SHIFT_TOKCLASS^ expr_additive )*
3913.   ;
3914.
3915. expr_additive :
3916.   expr_multiplicative ( ADDITIVE_TOKCLASS^ expr_multiplicative )*
3917.   ;
3918.
3919. expr_multiplicative :
3920.   expr_pm ( MULTIPLICATIVE_TOKCLASS^ expr_pm )*
3921.   ;
3922.
3923. expr_pm :
3924.   expr_unary ( PM_TOKCLASS^ expr_unary )*
3925.   ;
3926.
3927. expr_unary :
3928.   (
3929.     ( expr_postfix )?
3930.   |
3931.     tok0:BOOLEANUNARYOP_TOKCLASS^ exp0:expr_unary
3932.     << #0=#(#[TOK_BOOLEANUNARYOP, tok0->getText()], #exp0); >>
3933.   |
3934.     tok1:BOOLEANEVENTOP_TOKCLASS^ exp1:expr_unary
3935.     << #0=#(#[TOK_BOOLEANEVENTOP, tok1->getText()], #exp1); >>
3936.   |
3937.     tok2:NUMERICUNARYOP_TOKCLASS^ exp2:expr_unary
3938.     << #0=#(#[TOK_NUMERICUNARYOP, tok2->getText()], #exp2); >>
3939.   )
3940.   ;
```

```
3941.expr_postfix! :
3942.  exp0:expr_primary
3943.  << #0=#exp0; >>
3944.  (
3945.    (
3946.      tok0:TOK_LSQUARE exp1:expression TOK_RSQUARE
3947.      << #0=#(#[TOK_POSTOP, tok0->getText()], #0, #exp1); >>
3948.    |
3949.      tok1:TOK_LPAREN { exp2:expression_list } TOK_RPAREN
3950.      << #0=#(#[TOK_POSTOP, tok1->getText()], #0, #exp2); >>
3951.    |
3952.      tok2:TOK_DOT exp3:expr_id
3953.      << #0=#(#[TOK_POSTOP, tok2->getText()], #0, #exp3); >>
3954.    |
3955.      tok3:TOK_POINTERTO exp4:expr_id
3956.      << #0=#(#[TOK_POSTOP, tok3->getText()], #0, #exp4); >>
3957.    |
3958.      tok4:TOK_PLUSPLUS
3959.      << #0=#(#[TOK_POSTOP, tok4->getText()], #0); >>
3960.    |
3961.      tok5:TOK_MINUSMINUS
3962.      << #0=#(#[TOK_POSTOP, tok5->getText()], #0); >>
3963.    )
3964.  )*
3965.  ;
3966.
3967.expr_primary :
3968.    TOK_THIS
3969.  |
3970.    expr_id
3971.  |
3972.    expr_literal
3973.  |
3974.    TOK_LPAREN^ { expression } TOK_RPAREN
3975.  ;
3976.
3977.expr_id! :
3978.  << bool scope = false; >>
3979.  {
3980.    tok0:TOK_SCOPE
3981.    <<
3982.      #0=#[TOK_ID, tok0->getText()];
3983.      scope = true;
3984.    >>
3985.  }
3986.  tok1:TOK_ID
3987.  <<
3988.    if (!scope)
3989.      #0=#[TOK_ID, tok1->getText()];
3990.    else
3991.      strncat(((AST*)#0)->getText(), tok1->getText(), strlen(tok1->getText()));
3992.  >>
3993.  (
3994.    tok2:TOK_SCOPE
3995.    tok3:TOK_ID
3996.    <<
3997.      strncat(((AST*)#0)->getText(), tok2->getText(), strlen(tok2->getText()));
3998.      strncat(((AST*)#0)->getText(), tok3->getText(), strlen(tok3->getText()));
3999.    >>
4000.  )*
4001.  ;
```

```
4002. expr_literal :
4003.    constant
4004.  |
4005.    expr_string
4006.  ;
4007.
4008. constant :
4009.    CONSTANTNUMBER_TOKCLASS
4010.  |
4011.    TOK_CHARACTER
4012.  |
4013.    CONSTANTBOOL_TOKCLASS
4014.  ;
4015.
4016. expr_string! :
4017.  tok0:TOK_STRING
4018.  << #0=#[tok0->getType(), tok0->getText()]; >>
4019.  (
4020.    tok1:TOK_STRING
4021.    << strncat(((AST*)#0)->getText(), tok1->getText(), strlen(tok1->getText())); >>
4022.  )*
4023.  ;
4024. }
```

Anexo E. Gramática SORCERER

Neste anexo móstranse as gramáticas utilizadas para a xeración automática, coa aplicación Sorcerer [135], de analizadores/traductores das ASTs das expresións C++ estendidas cos operadores de evento e temporización. A primeira gramática serve para xerar un analizador/traductor que substitúa as expresións numéricas por variábeis booleanas nas expresións con eventos complexas, para a súa posterior reducción coa aplicación Sidoni, tal e como se explica en (§6.5.1.5.3). A segunda gramática xera un analizador/traductor que imprime as ASTs modificadas na sintaxe C++.

Substitución de expresións numéricas

```
4025. /*
4026.  *  Gramática C++ para SORCERER
4027.  *  Expresións C++ sen:
4028.  *    . especializacións de "templates"
4029.  *    . "casts"
4030.  *    . conversións de tipo explícitas
4031.  *    . operadores sizeof e typeid
4032.  *    . funcións membro operator
4033.  *    . funcións membro de conversión
4034.  *    . operadores de reserva e liberación de memoria
4035.  *    . modificadores de Microsoft nos nomes cualificados
4036.  *  Engadíronse:
4037.  *    . operadores de evento: UP_EDGE e DOWN_EDGE
4038.  */
4039. #header
4040. <<
4041.  #include <deque>
4042.  #include <set>
4043.  #include <string>
4044.  #include <sstream>
4045.  using namespace std;
4046.  #include "tokens.h"
4047.  #include "AToken.h"
4048.  #include "ATokenBuffer.h"
4049.  typedef ANTLRCommonToken ANTLRToken;
4050.  #include "ASTBase.h"
4051.  #include "ATokPtr.h"
4052.  #include "AST.h"
4053.  typedef AST SORAST;
4054.  #include "CPPTreePrinter.h"
4055. >>
```

```
4056. // DEFINICIÓN DO ANALIZADOR SINTÁCTICO DE ASTs
4057.
4058. class CPPTreeParser {
4059.
4060. <<
4061.  private:
4062.   astdata_seq* trees;
4063.   unsigned expr_number;
4064.   std::string add_expression(PCCTS_AST* tree, bool boolean = false);
4065.   void erase_ast(std::string name);
4066.   void reduce_expressions();
4067.  public:
4068.   void parse(SORASTBase **_root,
4069.              SORASTBase **_result,
4070.              astdata_seq& expressions,
4071.              unsigned& init_number);
4072. >>
4073.
4074. // GRAMÁTICA
4075.
4076. // Expresións xerais
4077.
4078. sfcpp_expression :
4079.   expr_boolean
4080.  |
4081.   exp0:expr_numerical
4082.   <<
4083.    std::string expr_name = add_expression(exp0);
4084.    #sfcpp_expression=#(#[TOK_NUMERICEXPR,
4085.                         const_cast<char*>(expr_name.c_str())],
4086.                         exp0);
4087.   >>
4088.  |
4089.   #( TOK_COMMA sfcpp_expression sfcpp_expression )
4090.  |
4091.   #( TOK_LPAREN sfcpp_expression TOK_RPAREN )
4092.  |
4093.   TOK_CHARACTER
4094.  |
4095.   TOK_STRING
4096.  |
4097.   TOK_ID
4098.  |
4099.   TOK_THIS
4100.  ;
4101.
4102. // Expresións booleanas
4103.
4104. expr_boolean :
4105.   #( TOK_BOOLEANUNARYOP sfcpp_expression )
4106.  |
4107.   #( tok0:TOK_BOOLEANEVENTOP exp0:sfcpp_expression )
4108.   <<
4109.    PCCTS_AST* aux = #(#[(ANTLRTokenType) tok0->type(), tok0->getText()], exp0);
4110.    std::string expr_name = add_expression(aux, true);
4111.    #expr_boolean=#(#[TOK_EVENTEXPR, const_cast<char*>(expr_name.c_str())], aux);
4112.   >>
4113.  |
4114.   expr_boolean_binary
4115.  |
4116.   TOK_BOOLTRUE
4117.  |
4118.   TOK_BOOLFALSE
4119.  ;
```

```
4120. expr_boolean_binary :
4121.    #( TOK_OR sfcpp_expression sfcpp_expression )
4122.  |
4123.    #( TOK_AND sfcpp_expression  sfcpp_expression )
4124.  ;
4125.
4126. // Expresións numéricas
4127.
4128. expr_numerical :
4129.    #( TOK_NUMERICUNARYOP sfcpp_expression )
4130.  |
4131.    expr_numerical_binary
4132.  |
4133.    #( TOK_QUESTIONMARK sfcpp_expression sfcpp_expression TOK_COLON sfcpp_expression )
4134.  |
4135.    #( TOK_POSTOP sfcpp_expression {sfcpp_expression} )
4136.  |
4137.    TOK_OCTALINT
4138.  |
4139.    TOK_DECIMALINT
4140.  |
4141.    TOK_HEXADECIMALINT
4142.  |
4143.    TOK_FLOATONE
4144.  |
4145.    TOK_FLOATTWO
4146.  ;
4147.
4148. expr_numerical_binary :
4149.    #( TOK_ASSIGNEQUAL sfcpp_expression sfcpp_expression )
4150.  |
4151.    #( TOK_TIMESEQUAL sfcpp_expression sfcpp_expression )
4152.  |
4153.    #( TOK_DIVIDEEQUAL sfcpp_expression sfcpp_expression )
4154.  |
4155.    #( TOK_MODEQUAL sfcpp_expression sfcpp_expression )
4156.  |
4157.    #( TOK_PLUSEQUAL sfcpp_expression sfcpp_expression )
4158.  |
4159.    #( TOK_BITWISEXOREQUAL sfcpp_expression sfcpp_expression )
4160.  |
4161.    #( TOK_MINUSEQUAL sfcpp_expression sfcpp_expression )
4162.  |
4163.    #( TOK_SHIFTLEFTEQUAL sfcpp_expression sfcpp_expression )
4164.  |
4165.    #( TOK_SHIFTRIGHTEQUAL sfcpp_expression sfcpp_expression )
4166.  |
4167.    #( TOK_BITWISEANDEQUAL sfcpp_expression sfcpp_expression )
4168.  |
4169.    #( TOK_BITWISEOREQUAL sfcpp_expression sfcpp_expression )
4170.  |
4171.    #( TOK_PLUS sfcpp_expression sfcpp_expression )
4172.  |
4173.    #( TOK_MINUS sfcpp_expression sfcpp_expression )
4174.  |
4175.    #( TOK_NOTEQUAL sfcpp_expression sfcpp_expression )
4176.  |
4177.    #( TOK_EQUAL sfcpp_expression sfcpp_expression )
4178.  |
4179.    #( TOK_STAR sfcpp_expression sfcpp_expression )
4180.  |
4181.    #( TOK_DIVIDE sfcpp_expression sfcpp_expression )
4182.  |
4183.    #( TOK_MOD sfcpp_expression sfcpp_expression )
4184.  |
```

```
4185.    #( TOK_LESSTHAN sfcpp_expression sfcpp_expression )
4186.  |
4187.    #( TOK_GREATERTHAN sfcpp_expression sfcpp_expression )
4188.  |
4189.    #( TOK_LESSTHANOREQUALTO sfcpp_expression sfcpp_expression )
4190.  |
4191.    #( TOK_GREATERTHANOREQUALTO sfcpp_expression sfcpp_expression )
4192.  |
4193.    #( TOK_SHIFTLEFT sfcpp_expression sfcpp_expression )
4194.  |
4195.    #( TOK_SHIFTRIGHT sfcpp_expression sfcpp_expression )
4196.  ;
4197.}
4198.
4199.// MÉTODOS DO ANALIZADOR
4200.
4201.<<
4202.  std::string CPPTreeParser::add_expression(PCCTS_AST* tree, bool boolean)
4203.  {
4204.    char num[5];
4205.    itoa(expr_number++, num, 10);
4206.    ast_data data;
4207.    data.name = "sfcpp_expr";
4208.    data.name.append(num);
4209.    data.type = boolean;
4210.    data.ast = tree;
4211.    trees->push_back(data);
4212.    return data.name;
4213.  }
4214.
4215.  void CPPTreeParser::erase_ast(std::string name)
4216.  {
4217.    for(astdata_iterator it = trees->begin(); it != trees->end(); ++it)
4218.      if (it->name == name)
4219.      {
4220.        trees->erase(it);
4221.        return;
4222.      }
4223.  }
4224.
4225.  void CPPTreeParser::reduce_expressions()
4226.  {
4227.    std::set<std::string> names;
4228.    // eliminar expresións numéricas contidas noutras expresións numéricas
4229.    for(const_astdata_iterator it = trees->begin(); it != trees->end(); ++it)
4230.    {
4231.      if (!it->type)
4232.      {
4233.        ((AST*) it->ast)->find_numexpr(names);
4234.        for(std::set<std::string>::const_iterator iter = names.begin();
4235.            iter != names.end();
4236.            ++iter)
4237.          erase_ast(*iter);
4238.        names.clear();
4239.      }
4240.    }
4241.    for(it = trees->begin(); it != trees->end(); ++it)
4242.      names.insert(it->name);
4243.    for(it = trees->begin(); it != trees->end(); ++it)
4244.      for(std::set<std::string>::const_iterator iter = names.begin();
4245.          iter != names.end();
4246.          ++iter)
4247.        if (it->name != *iter)
4248.          ((AST*) it->ast)->remove_expr(*iter);
4249.  }
```

```
4250.  void CPPTreeParser::parse(SORASTBase **_root,
4251.                           SORASTBase **_result,
4252.                           astdata_seq& expressions,
4253.                           unsigned& init_number)
4254.  {
4255.   trees = &expressions;
4256.   expr_number = init_number;
4257.   // obter expresións
4258.   sfcpp_expression(_root, _result);
4259.   init_number = expr_number;
4260.   // reducir expresións
4261.   std::set<std::string> names;
4262.   ((AST*)(*_result))->find_eventexpr(names);
4263.   for(std::set<std::string>::const_iterator iter = names.begin();
4264.       iter != names.end();
4265.       ++iter)
4266.    erase_ast(*iter);
4267.   reduce_expressions();
4268.   for(astdata_iterator it = trees->begin(); it != trees->end(); ++it)
4269.    ((AST*)(*_result))->remove_expr(it->name);
4270.   // producir resultado
4271.   CPPTreePrinter tree_printer;
4272.   for(it = trees->begin(); it != trees->end(); ++it)
4273.   {
4274.    stringstream output;
4275.    tree_printer.dump(&it->ast, output);
4276.    it->code = output.str();
4277.   }
4278.  }
4279. >>
```

Impresión ASTs na sintaxe C++

```
4280. /*
4281.  *  Gramática C++ para SORCERER: impresión das ASTs
4282.  *  Expresións C++ sen:
4283.  *    . especializacións de "templates"
4284.  *    . "casts"
4285.  *    . conversións de tipo explícitas
4286.  *    . operadores sizeof e typeid
4287.  *    . funcións membro operator
4288.  *    . funcións membro de conversión
4289.  *    . operadores de reserva e liberación de memoria
4290.  *    . modificadores de Microsoft nos nomes cualificados
4291.  *  Engadíronse:
4292.  *    . operadores de evento: UP_EDGE e DOWN_EDGE
4293.  */
4294. #header
4295. <<
4296.  #include <deque>
4297.  #include <iostream>
4298.  #include <string>
4299.  #include <set>
4300.  using namespace std;
4301.  #include "tokens.h"
4302.  #include "AToken.h"
4303.  #include "ATokenBuffer.h"
4304.  typedef ANTLRCommonToken ANTLRToken;
4305.  #include "ASTBase.h"
4306.  #include "ATokPtr.h"
4307.  #include "AST.h"
4308.  typedef AST SORAST;
4309. >>
```

```
4310. // DEFINICIÓN DO IMPRESOR DE ASTs
4311.
4312. class CPPTreePrinter {
4313.
4314. <<
4315. private:
4316.   ostream* out;
4317. public:
4318.   void dump(SORASTBase** _root, ostream& out_stream = cout);
4319. >>
4320.
4321. // GRAMÁTICA
4322.
4323. // Expresións xerais
4324.
4325. sfcpp_expression :
4326.     expr_boolean
4327.   |
4328.     expr_numerical
4329.   |
4330.     #(
4331.       tok0:TOK_COMMA
4332.       sfcpp_expression
4333.       << (*out) << tok0->getText(); >>
4334.       sfcpp_expression
4335.     )
4336.   |
4337.     #(
4338.       tok1:TOK_LPAREN
4339.       << (*out) << tok1->getText(); >>
4340.       sfcpp_expression
4341.       tok2:TOK_RPAREN
4342.       << (*out) << tok2->getText(); >>
4343.     )
4344.   |
4345.     tok3:TOK_CHARACTER
4346.     << (*out) << tok3->getText(); >>
4347.   |
4348.     tok4:TOK_STRING
4349.     << (*out) << tok4->getText(); >>
4350.   |
4351.     tok5:TOK_ID
4352.     << (*out) << tok5->getText(); >>
4353.   |
4354.     tok6:TOK_THIS
4355.     << (*out) << tok6->getText(); >>
4356.   ;
4357.
4358. // Expresións booleanas
4359.
4360. expr_boolean :
4361.     <<  bool expr = false;  >>
4362.     #(
4363.       tok0:TOK_EVENTEXPR
4364.       {
4365.         expr_boolean
4366.         <<  expr = true;  >>
4367.       }
4368.     )
4369.     <<
4370.       if (!expr)
4371.         (*out) << tok0->getText();
4372.     >>
4373.   |
```

```
4374.    #(
4375.     tok1:TOK_BOOLEANUNARYOP
4376.     << (*out) << tok1->getText(); >>
4377.     sfcpp_expression
4378.    )
4379.   |
4380.    #(
4381.     tok2:TOK_BOOLEANEVENTOP
4382.     << (*out) << tok2->getText(); >>
4383.     sfcpp_expression
4384.    )
4385.   |
4386.    expr_boolean_binary
4387.   |
4388.    TOK_BOOLTRUE
4389.    << (*out) << "(1)";  >>
4390.   |
4391.    TOK_BOOLFALSE
4392.    << (*out) << "(0)";  >>
4393.   ;
4394.
4395. expr_boolean_binary :
4396.    #(
4397.     tok0:TOK_OR
4398.     sfcpp_expression
4399.     << (*out) << tok0->getText(); >>
4400.     sfcpp_expression
4401.    )
4402.   |
4403.    #(
4404.     tok1:TOK_AND
4405.     sfcpp_expression
4406.     << (*out) << tok1->getText(); >>
4407.     sfcpp_expression
4408.    )
4409.   ;
4410.
4411. // Expresións numéricas
4412.
4413. expr_numerical :
4414.    <<  bool expr = false;  >>
4415.    #(
4416.     tok0:TOK_NUMERICEXPR
4417.     {
4418.      expr_numerical
4419.      << expr = true;  >>
4420.     }
4421.    )
4422.    <<
4423.     if (!expr)
4424.      (*out) << tok0->getText();
4425.    >>
4426.   |
4427.    #(
4428.     tok1:TOK_NUMERICUNARYOP
4429.     << (*out) << tok1->getText(); >>
4430.     sfcpp_expression
4431.    )
4432.   |
4433.    expr_numerical_binary
4434.   |
4435.    #(
4436.     tok2:TOK_QUESTIONMARK
4437.     sfcpp_expression
4438.     << (*out) << tok2->getText(); >>
4439.     sfcpp_expression
```

```
4440.     tok3:TOK_COLON
4441.     << (*out) << tok3->getText(); >>
4442.     sfcpp_expression
4443.    )
4444.  |
4445.    #(
4446.     tok4:TOK_POSTOP
4447.     sfcpp_expression
4448.     <<
4449.      char* text = tok4->getText();
4450.      (*out) << text;
4451.     >>
4452.     {sfcpp_expression}
4453.     <<
4454.      if (text[0] == '[')
4455.        (*out) << "]";
4456.      else if (text[0] == '(')
4457.        (*out) << ")";
4458.     >>
4459.    )
4460.  |
4461.    tok5:TOK_OCTALINT
4462.    << (*out) << tok5->getText(); >>
4463.  |
4464.    tok6:TOK_DECIMALINT
4465.    << (*out) << tok6->getText(); >>
4466.  |
4467.    tok7:TOK_HEXADECIMALINT
4468.    << (*out) << tok7->getText(); >>
4469.  |
4470.    tok8:TOK_FLOATONE
4471.    << (*out) << tok8->getText(); >>
4472.  |
4473.    tok9:TOK_FLOATTWO
4474.    << (*out) << tok9->getText(); >>
4475.  ;
4476.
4477.expr_numerical_binary :
4478.    #(
4479.     tok0:TOK_MODEQUAL
4480.     sfcpp_expression
4481.     << (*out) << tok0->getText(); >>
4482.     sfcpp_expression
4483.    )
4484.  |
4485.    #(
4486.     tok1:TOK_ASSIGNEQUAL
4487.     sfcpp_expression
4488.     << (*out) << tok1->getText(); >>
4489.     sfcpp_expression
4490.    )
4491.  |
4492.    #(
4493.     tok2:TOK_TIMESEQUAL
4494.     sfcpp_expression
4495.     << (*out) << tok2->getText(); >>
4496.     sfcpp_expression
4497.    )
4498.  |
4499.    #(
4500.     tok3:TOK_DIVIDEEQUAL
4501.     sfcpp_expression
4502.     << (*out) << tok3->getText(); >>
4503.     sfcpp_expression
4504.    )
```

```
4505.  |
4506.   #(
4507.    tok4:TOK_PLUSEQUAL
4508.    sfcpp_expression
4509.    << (*out) << tok4->getText(); >>
4510.    sfcpp_expression
4511.   )
4512.  |
4513.   #(
4514.    tok5:TOK_BITWISEXOREQUAL
4515.    sfcpp_expression
4516.    << (*out) << tok5->getText(); >>
4517.    sfcpp_expression
4518.   )
4519.  |
4520.   #(
4521.    tok6:TOK_MINUSEQUAL
4522.    sfcpp_expression
4523.    << (*out) << tok6->getText();  >>
4524.    sfcpp_expression
4525.   )
4526.  |
4527.   #(
4528.    tok7:TOK_SHIFTLEFTEQUAL
4529.    sfcpp_expression
4530.    << (*out) << tok7->getText();  >>
4531.    sfcpp_expression
4532.   )
4533.  |
4534.   #(
4535.    tok8:TOK_SHIFTRIGHTEQUAL
4536.    sfcpp_expression
4537.    << (*out) << tok8->getText(); >>
4538.    sfcpp_expression
4539.   )
4540.  |
4541.   #(
4542.    tok9:TOK_BITWISEANDEQUAL
4543.    sfcpp_expression
4544.    << (*out) << tok9->getText(); >>
4545.    sfcpp_expression
4546.   )
4547.  |
4548.   #(
4549.    tok10:TOK_BITWISEOREQUAL
4550.    sfcpp_expression
4551.    << (*out) << tok10->getText(); >>
4552.    sfcpp_expression
4553.   )
4554.  |
4555.   #(
4556.    tok11:TOK_PLUS
4557.    sfcpp_expression
4558.    << (*out) << tok11->getText(); >>
4559.    sfcpp_expression
4560.   )
4561.  |
4562.   #(
4563.    tok12:TOK_MINUS
4564.    sfcpp_expression
4565.    << (*out) << tok12->getText(); >>
4566.    sfcpp_expression
4567.   )
4568.  |
```

```
4569.   #(
4570.    tok13:TOK_NOTEQUAL
4571.    sfcpp_expression
4572.    << (*out) << tok13->getText(); >>
4573.    sfcpp_expression
4574.   )
4575.  |
4576.   #(
4577.    tok14:TOK_EQUAL
4578.    sfcpp_expression
4579.    << (*out) << tok14->getText(); >>
4580.    sfcpp_expression
4581.   )
4582.  |
4583.   #(
4584.    tok15:TOK_STAR
4585.    sfcpp_expression
4586.    << (*out) << tok15->getText();  >>
4587.    sfcpp_expression
4588.   )
4589.  |
4590.   #(
4591.    tok16:TOK_DIVIDE
4592.    sfcpp_expression
4593.    << (*out) << tok16->getText(); >>
4594.    sfcpp_expression
4595.   )
4596.  |
4597.   #(
4598.    tok17:TOK_MOD
4599.    sfcpp_expression
4600.    << (*out) << tok17->getText(); >>
4601.    sfcpp_expression
4602.   )
4603.  |
4604.   #(
4605.    tok18:TOK_LESSTHAN
4606.    sfcpp_expression
4607.    << (*out) << tok18->getText(); >>
4608.    sfcpp_expression
4609.   )
4610.  |
4611.   #(
4612.    tok19:TOK_GREATERTHAN
4613.    sfcpp_expression
4614.    << (*out) << tok19->getText(); >>
4615.    sfcpp_expression
4616.   )
4617.  |
4618.   #(
4619.    tok20:TOK_LESSTHANOREQUALTO
4620.    sfcpp_expression
4621.    << (*out) << tok20->getText(); >>
4622.    sfcpp_expression
4623.   )
4624.  |
4625.   #(
4626.    tok21:TOK_GREATERTHANOREQUALTO
4627.    sfcpp_expression
4628.    << (*out) << tok21->getText(); >>
4629.    sfcpp_expression
4630.   )
4631.  |
```

```
4632.    #(
4633.     tok22:TOK_SHIFTLEFT
4634.     sfcpp_expression
4635.     << (*out) << tok22->getText(); >>
4636.     sfcpp_expression
4637.    )
4638.  |
4639.    #(
4640.     tok23:TOK_SHIFTRIGHT
4641.     sfcpp_expression
4642.     << (*out) << tok23->getText(); >>
4643.     sfcpp_expression
4644.    )
4645.  ;
4646.}
4647.
4648.// MÉTODOS DO IMPRESOR
4649.
4650.<<
4651.  void CPPTreePrinter::dump(SORASTBase** _root, ostream& out_stream)
4652.  {
4653.    out = &out_stream;
4654.    sfcpp_expression(_root);
4655.  }
4656.>>
```

Anexo F. Acceso remoto á máquina virtual

Neste anexo inclúense os códigos das mensaxes, indícanse os patróns de comunicación e móstrase o intercambio de mensaxes utilizados para acceder aos servicios remotos da máquina virtual (§7.4.3).

F.1. Códigos das mensaxes

A Táboa F-I mostra, clasificados por categorías, os códigos das mensaxes (§7.4.1) utilizadas na implementación actual dos diferentes servicios de acceso remoto da máquina virtual.

Categoría	Códigos
Carga e descarga de DLLs	LOAD_DLL UNLOAD_DLL UNLOAD_ALL_DLL
Carga e descarga de táboas de E/S	LOAD_SYSTEMIO UNLOAD_SYSTEMIO
Carga, descarga e execución de modelos	LOAD_MODEL UNLOAD_MODEL PLAY_MODEL NEXT_MODEL_EVOLUTION STOP_MODEL
Xestión de configuracións	ADD_CONF REMOVE_CONF SET_CURRENT_CONFIGURATION GET_CURRENT_CONFIGURATION GET_CONFIGURATION_INFO ACTIVATE DEACTIVATE
Consulta da estuctura de módulos da máquina virtual	GET_VM_INFO GET_MODULE_INFO
Consulta da información de configuración dos "drivers" de E/S	GET_DEVICE_INFO
Desconexión e finalización da execución da máquina virtual	DISCONNECT SHUTDOWN

Táboa F-I. Códigos das mensaxes dos servicios de acceso remoto da máquina virtual.

No resto deste anexo descríbense os intercambios de mensaxes de cada servicio e indícase o patrón de comunicacións (§7.4.2) utilizado para cada un.

F.2. Carga e descarga de DLLs

En todos os servicios desta categoría utilízase unha orde simple sen resposta (patrón 1). O intercambio de mensaxes en cada caso concreto é o seguinte:

F.2.1. Carga dunha DLL

Na máquina virtual poden almacenarse simultaneamente múltiples DLLs, polo que ademais de indicarse a localización da DLL, proporciónase tamén como parámetro desta orde un valor alfanumérico único que será utilizado para identificar a DLL.

⇨ LOAD_DLL, 0, 2, *dll_name*, *dll_path*
⇦ OK!

F.2.2. Descarga dunha DLL

⇨ UNLOAD_DLL, 0, 1, *dll_name*
⇦ OK!

F.2.3. Descarga de todas as DLLs

⇨ UNLOAD_ALL_DLL, 0, 0
⇦ OK!

F.3. Carga e descarga de táboas de E/S

En todos os servicios desta categoría utilízase unha orde simple sen resposta (patrón 1). O intercambio de mensaxes en cada caso concreto é o seguinte:

F.3.1. Carga dunha táboa de E/S

A versión actual da máquina virtual non permite cargar máis dunha táboa de E/S simultaneamente na memoria, polo que unicamente se pasa como parámetro da orde o directorio no que está localizada a táboa.

⇨ LOAD_SYSTEMIO, 0, 1, *io_table_path*
⇦ OK!

F.3.2. Descarga dunha táboa de E/S

⇨ UNLOAD_SYSTEMIO, 0, 0
⇦ OK!

F.4. Carga, descarga e execución de modelos

En todos os servicios desta categoría utilízase para as comunicacións unha orde simple sen resposta (patrón 1) excepto na continuación da execución dun modelo, no que se recibe unha

resposta formada por múltiples mensaxes (patrón 2) que conteñen a información utilizada para a depuración da execución do modelo. O intercambio de mensaxes en cada caso concreto é o seguinte:

F.4.1. Carga dun modelo

⇨ LOAD_MODEL, 0, 2, *module_name*, *model_name*
⇦ OK!

F.4.2. Descarga dun modelo

⇨ UNLOAD_MODEL, 0, 2, *module_name*, *model_name*
⇦ OK!

F.4.3. Inicio da execución dun modelo

O terceiro parámetro desta orde contén a información que permite configurar o inicio da execución do modelo. Na versión actual este parámetro é utilizado para indicar dous tipos de opcións:

1. As relacionadas coa configuración do intérprete Grafcet (§8.3.2): ¿Cómo son considerados os eventos externos no tempo interno?. ¿Os eventos internos son considerados na seguinte evolución interna?. ¿Permítese a utilización de accións internas?. ¿Os temporizadores utilizan unha semántica redisparábel (§8.6.2)?
2. A activación ou non do modo de depuración.

⇨ PLAY_MODEL, 0, 3, *module_name*, *model_name*, *options*
⇦ OK!

F.4.4. Detención da execución dun modelo

⇨ STOP_MODEL, 0, 2, *module_name*, *model_name*
⇦ OK!

F.4.5. Continuación da execución dun modelo

Na execución dos modelos en modo depuración, despois de cada evolución, a máquina virtual envía a información sobre a situación e os valores das variábeis do modelo. No caso dos modelos Grafcet a situación envíase nunha mensaxe que contén os identificadores alfanuméricos das etapas activas, e os valores das variábeis envíanse en formato binario dividido en varias secuencias que o receptor ten que concatenar e interpretar.

⇨ NEXT_MODEL_EVOLUTION, 0, 2, *module_name*, *model_name*
⇦ OK, 1, 2, *module_name*, *model_name*
⇦ OK, 1, situation_size, *active_step_1*, *active_step_2*, ... , *active_step_N*
⇦ OK, 1, model_info_size, *model_info_1*, *mode_info_2*, ... ,*model_info_N*
⇦ OK!

F.5. Xestión de configuracións.

No servicio para engadir unha configuración utilízase unha orde múltiple sen resposta (patrón 3), nos servicios de consulta unha orde simple con resposta (patrón 2), e en todos os demais unha orde simple sen resposta (patrón 1). O intercambio de mensaxes en cada caso concreto é o seguinte:

F.5.1. Engadir unha configuración

A configuración está formada por un conxunto de módulos agrupados en categorías dacordo ao seu tipo. O cliente envía dúas ordes por cada categoría, a primeira contén o nome da categoría e o número máximo de módulos permitidos nela; e o segundo a lista de módulos da categoría.

⇨ ADD_CONF, 1, 2, *module_name*, *configuration_name*
⇦ OK!
// Repetir para cada categoría da configuración
⇨ ADD_CONF, modules_max_size, 1, *category_name*
⇦ OK!
⇨ ADD_CONF, 1, modules_size, *module_name_1*, ... , *module_name_N*
⇦ OK!
⇨ ADD_CONF, 0, 0
⇦ OK!

F.5.2. Eliminar unha configuración

⇨ REMOVE_CONF, 0, 2, *module_name*, *configuration_name*
⇦ OK!

F.5.3. Activar unha configuración

⇨ SET_CURRENT_CONFIGURATION, 0, 2, *module_name*, *configuration_name*
⇦ OK!

F.5.4. Consultar a configuración activa

⇨ GET_CURRENT_CONFIGURATION, 0, 1, *module_name*
⇦ OK, 1, 1, *configuration_name*
// Repetir para cada categoría da configuración
⇦ OK, modules_max_size, 1, *category_name*
⇦ OK, 1, modules_size, *module_name_1*, ... , *module_name_N*
⇦ OK!

F.5.5. Consultar as configuracións dispoñíbeis

⇨ GET_CONFIGURATION_INFO, 0, 1, *module_name*
  // Repetir para cada configuración
  ⇦ OK, 1, 1, *configuration_name*
    // Repetir para cada categoría da configuración
    ⇦ OK, modules_max_size, 1, *category_name*
    ⇦ OK, 1, modules_size, *module_name_1, ... , module_name_N*
⇦ OK!

F.5.6. Activar un módulo nunha configuración

Activar un módulo na configuración dun módulo da máquina virtual. Os parámetros desta mensaxe son o nome do módulo da máquina virtual, o nome da configuración dese módulo que vai ser modificada, e o tipo e o nome do módulo a activar.

⇨ ACTIVATE, 0, 4, *module_name, configuration_name, type_name, module_name*
⇦ OK!

F.5.7. Desactivar un módulo nunha configuración

⇨ DEACTIVATE, 0, 4, *module_name, configuration_name, type_name, module_name*
⇦ OK!

F.6. Consulta da estructura de módulos da máquina virtual.

En todos os servicios desta categoría utilízase unha orde simple con resposta (patrón 2). O intercambio de mensaxes en cada caso concreto é o seguinte:

F.6.1. Consultar a estructura da máquina virtual

Este servicio devolve información sobre a estructura de módulos que forma a máquina virtual. Para cada módulo M (incluída a propia máquina) recíbese unha mensaxe de resposta contendo o nome de M, o nome do módulo utilizado como almacén dos módulos cos que se forma a estructura lóxica de M, e a lista de módulos agregados que forman a estructura de M no momento de executar o servicio.

⇨ GET_VM_INFO, 0, 0
⇦ OK, 1, 1, *vmachine_name*
  // Repetir para cada módulo
  ⇦ OK, 1, parts_size+2, *module_name, module_store_name, part_1, ... , part_N*
⇦ OK!

F.6.2. Consultar os módulos dispoñíbeis nun almacén de módulos

Este servicio devolve para cada categoría dos módulos almacenados no almacén indicado como parámetro da orde, dúas mensaxes: a primeira co nome da categoría e, a segunda, coa

lista de módulos desa categoría presentes no almacén e o nome da DLL dende a que se cargou cada un deles na memoria da máquina virtual.

⇨ GET_MODULE_INFO, 0, 1, *module_name*
  // Repetir para cada tipo de módulo
  ⇦ OK, 1, 1, *type_name*
  ⇦ OK, 1, modules_size*2, *module_1, dll_1, ... , module_N, dll_N*
⇦ OK!

F.7. Consulta da información de configuración dos "drivers" de E/S

En todos os servicios desta categoría utilízase unha orde simple con resposta (patrón 2). O intercambio de mensaxes é o seguinte:

F.7.1. Consulta da información de configuración dun "driver" de E/S

Na versión actual unicamente se envía a información da estructura do dispositivo e non se inclúen as características, valores e condicións de validez.

⇨ GET_DEVICE_INFO, 0, 1, *driver_name*
⇦ OK, num_levels, 1, *device_name*
  // Obter información do nivel principal
  ⇦ OK, 1, 4, *main_level_name, main_level_type, main_level_order, subsystems_size*
   // Repetir para cada subsistema
   ⇦ OK, num_levels, 1
     // Repetir para cada nivel do subsistema
     ⇦ OK, 1, 4, *level_name, level_type, level_order, subsystems_size*
     // Repetir recursivamente para cada subsistema
     ...
⇦ OK!

F.8. Desconexión e finalización da execución da máquina virtual

En todos os servicios desta categoría utilízase unha orde simple sen resposta (patrón 1). O intercambio de mensaxes en cada caso concreto é o seguinte:

F.8.1. Desconexión do cliente

A versión actual da máquina virtual non permite máis dunha conexión simultánea.

⇨ DISCONNECT, 0, 0
⇦ OK!

F.8.2. Apagado da máquina virtual

⇨ SHUTDOWN, 0, 0
⇦ OK!

Bibliografía

[1] AFCET, Association Française des Sciences et Technologies de l'Information et des Systèmes, Groupe Systèmes Logiques. *Pour une représentation normalisée du cahier de charges dún automatisme logique.* In: Automatique et Informatique Industrielle, 61, pp: 27-32, 62, pp: 36-40, 1977.

[2] AFCET, Association Française des Sciences et Technologies de l'Information et des Systèmes, Groupe Systèmes Logiques. *Interprétations algébriques et algorithmiques et temporisations.* In: Le Nouvel Automatisme, 1983.

[3] Agha, G. *Actors: a model of concurrent computation in distributed systems.* Cambridge, MA: the MIT Press, 1986.

[4] AMICE. *CIMOSA: Open systems architecture for CIM*, 2nd version, Springer-Verlag, Berlin, 1993.

[5] Anderson, E. Bai, Z. Bischof, C. Demmel, J. Dongarra, J. Du Croz, J. Greenbaum, A. Hammarling, S. McKenney, A. Ostrouchov, S. and Sorensen, D. *LAPACK Users' Guide*, Release 1.0. SIAM, Philadelphia, 1992.

[6] André, C. & Peraldi, M.A. *Grafcet et langages synchrones.* In: Proceedings of the Grafcet'92 conference, pp. 90-100, Paris, France, 1992.

[7] André, C. & Gaffé, D. *Proving properties of Grafcet with synchronous tools.* In: Proceedings of IMACS/IEEE CESA'96 Multiconference, pp: 777-782, Lille, France, 1996.

[8] Antsaklis, P., Koutsoukos, X. & Zaytoon, J. *On hybrid control of complex systems: a survey.* In: 3rd International Conference ADMP'98, Automation of Mixed Processes, Dynamic Hybrid Systems, pp: 1-8, Reims, France, 1998.

[9] Arzén, K. *Grafcet for intelligent supervisory control applications.* In: Automatica, 30:10, 1994.

[10] Arzén, K. *Graphchart, a graphical language for sequential supervisory control.* In: Proc. of IFAC World Congress, pp. 407-412, San Francisco, 1996.

[11] Austern, M. *Generic Programming and the STL.* Addison-Wesley, 1998. ISBN 0-201-30956-4.

[12] Azzopardi D, Holding D J and Genovese G, *Object-Oriented Petri Net Synthesis for Modelling a Semiconductor Testing Plant.* In: Proceedings of IMACS/IEEE CESA'96 Multiconference, Symposium on Discrete Events and Manufacturing Systems, Lille, France, pp: 374-379, 1996.

[13] Aygalinc, P. & Denat, J.P. *Validation of functional Grafcet models and performance evaluation of the associated system using Petri Nets.* In: Automatic Control Production Systems, 27, pp: 81-93. 1993.

[14] Babb, M. *IEC 1131-3 seeks the attention of U.S. engineers.* In: Control Engineering Magazine, February, 1997.

[15] Baracos, Paul. *Automation Engineering in North America.* In: Proc. of 2nd Int. Conf. on Industrial Automation, pp: 3-8, Nancy, France, 1995.

[16] Baracos, Paul. *Grafcet: a North American perspective.* In: Proc. of 9th IFAC Symposium on Information Control in Manufacturing, pp: 41-46, Nancy-Metz, France, 1998. Elsevier Science Ltd. ISBN: 0-08-042928-9.

[17] Barker, H.A., Chen, M., Grant, P.W., Jobling, C.P. & Townsend, P. *Open Architecture for Computer-Aided Control Engineering.* In: IEEE Control Systems Magazine, 13(2), pp: 17-27, 1993.

[18] Bass, J.M., Schooling, S. & Turnbull, G. *Process Control Systems Integration.* In: Preprints of 8th IFAC symposium on Computer Aided Control Systems Design, Saldford, UK, 2000.

[19] Bauer, N. and Engell, S. *A comparison of Sequential Function Charts and Statecharts and an approach towards integration.* In: Proc. 2nd Int. Workshop on Integration of Specification Techniques for Applications in Engineering, pp: 58-69, Grenoble, France, 2002.

[20] Benner, P., Mehrmann, V., Sima, V., Van Huffel, S. and A. Varga. *SLICOT - A Subroutine Library in Systems and Control Theory*, NICONET Report 97-3, June 1997.

[21] Benveniste, A. & Berry, G. *The synchronous approach to reactive and real-time systems.* In: Proc. of the IEEE, 79(9), pp: 1270-1282, 1991.

[22] Benveniste, A., LeGuernic, P. and Jacquemot, Ch. *Synchronous programming with events and relations: the SIGNAL language and its semantics.* Science of Computer Programming, 16:103-149, 1991.

[23] Bernardi, S., Donatelli, S. & Merseguer, J. *From UML Sequence Diagrams and Statecharts to analysable Petri Net models.* In: Proc. of 3rd Int. Workshop on Software and Performance, pp: 35-45, Rome, Italy, 2002. ISBN:1-58113-563-7.

[24] Berry G. and Gonthier, G. *The ESTEREL synchronous language: design, semantics, and implementation.* In: Science of Computer Programming, 19(2), pp: 87-152, 1992.

[25] Bierel, E., Douchin, O. & Lhoste, P. *Grafcet: from theory to implementation.* In: Automatique, Productique et Informatique Industrielle, 31(3), pp: 543-559, 1997.

[26] Bilqees, A. *Computing Environments for Control Engineering.* PhD. University of Cambridge, 1996.

[27] Boissier, R., Dima, B., Razafindramary, D. and Soriano, T. *Hybrid systems modelling and validating using Statecharts and Grafcet.* In: Proc. of the IFAC Int. Workshop on Real Time Programming, pp: 73-9, 1994.

[28] Booch, G. *Análisis y diseño orientado a objetos con aplicaciones.* 2ª ed. Addison-Wesley. 1994. ISBN: 0-201-60122-2.

[29] Booch, G., Rumbaugh, J. & Jacobson, I. *El lenguaje unificado de modelado.* Ed. Addison-Wesley Iberoamericana, 1999. ISBN: 84-7829-028-1.

[30] Bouteille N, Brard P, Colombari G, Cotaina N & Richet D. *Le Grafcet*. 2ª ed. Cépaduès Éditions, 1995. ISBN: 2-85428-380-5.

[31] Brendel, W. & Tiegelkamp, M. *Uniform PLC programming in accordance with IEC 61131-3*. Infoteam Software GmbH, Bubenreuth, Germany.

[32] Buck, J. T. Ha, S. Lee, E. A. and Messerschmitt, D. G. *Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems*. In: Int. Journal of Computer Simulation, special issue on Simulation Software Development, 4, pp: 155-182, April, 1994.

[33] Cassez, Franck. *Formal semantics for reactive Grafcet*. In: Automatique, Productique et Informatique Industrielle, 31(3), pp: 581-603, 1997.

[34] CEN. *ENV 40 003: Computer-Integrated Manufacturing — Systems Architecture — Framework for Enterprise Modelling*, CEN/CENELEC, Brussels, 1990.

[35] Chacón E, Moreno W and Hennet J C, *Towards an Implementation of Hierarchical Hybrid Control Systems for the Integrated Operation of Industrial Complexes*. In: Proc. of IMACS/IEEE CESA'96 Multiconference, Symposium on Discrete Events and Manufacturing Systems, pp: 396-401, Lille, France, 1996.

[36] Chalmeta, R. *Reference Architecture for Enterprise Integration*. PhD. Universidad Politécnica de Valencia.

[37] Chapulart, V., Giaconne, T., Monneret, G., Pereyrol, F., Prunet, F. & Simottel, D. *A structured model for specification of Discrete Event Systems: the ACSY model*. In: Automatic Control Production Systems, 27, pp: 65-80. 1993.

[38] Chapulart, V., Larnac, M. & Dray, G. *Analysis and formal verification of Grafcet using interpreted sequential machine*. In: Proc. of IMACS-IEEE CESA'96 Multiconference, Symposium on Discrete Events and Manufacturing Systems, pp: 758-764, Lille, France, 1996.

[39] Char, B.W., Geddes, K.O., Gentleman, W.M. and Gonnet, G.H. *The design of Maple: A compact, portable, and powerful computer algebra system*. In: Lecture Notes in Computer Science, 162, pp. 101-115, Springer-Verlag, Berlin, 1983.

[40] Charbonnier, F., Alla, H. & David, R. *The supervised control of discrete event dynamic systems: a new approach*. In: Proc. of 34th Conf. on Decision and Control, New Orleans, USA, 1995.

[41] Chen, P. *The Entity-Relationship model: toward a unified view of data*. In: ACM Transactions on Database Systems, 1(1), pp: 9-36, 1976.

[42] Crater, K.C. *When technology standards become counterproductive*. In: Control Engineering Magazine, 1992.

[43] Dahl, O.J. & Nygaard, K. *SIMULA — An Algol Based Simulation Language*. In: Comm. ACM, 9(9), pp. 671-678, 1966.

[44] Damodar,Y.G. & Carol, L.S. *The just-in-time philosophy: a literature review*. In: Int. Journal of Production Research, 29(4), pp: 657-76, 1991.

[45] David R. *Grafcet: a powerful tool for specification of logic controllers*. In: IEEE transaction on control systems technology, 3, pp: 253-368, 1995.

[46] David, R. & Alla, H. *Du Grafcet aux Réseaux de Petri*. 2ª ed. Ed. Hermes, Paris, 1997. ISBN: 2-86601-325-5.

[47] Delatour, J. & Lamotte, F. *ArgoPN: a CASE tool merging UML and Petri Nets.* In: Proc. of 1st Int. Workshop on Validation and Verification of software for Enterprise Information Systems, pp: 94-102. ISBN: 972-98816-2-6.

[48] De Loor, P. *Du TTM/RTTL pour la validation des systèmes commandés par Grafcet.* PhD Thesis, University of Reims, France, 1996.

[49] Dima, Bruno & Manka, Marc. *Graf7-C version 2.1 DOS. Manuel de l'utilisateur, tutoriel et exemples Grafcet.* Ministère de l'éducation du Quebec, 1995. ISBN: 2-89740-009-1.

[50] Dongarra, J.J., Moler, C.B., Bunch J.R. & Stewart, G.W. *LINPACK Users' Guide.* 1979.

[51] Douglass, B. *Real-time design patterns: robust scalable architecture for real-time systems.* Addison-Wesley, 2003. ISBN: 0-201-69956-7.

[52] Douglass, B. *Doing hard time: developing real-time systems with UML, objects, frameworks and patterns.* Addison-Wesley, 1999. ISBN: 0-201-49837-5.

[53] Douglass, B. *Real-time design patterns: robust scalable architecture for real-time systems.* Addison-Wesley, 2003. ISBN: 0-201-69956-7.

[54] Doumeingts, G., Vallespir, B., Zanettin, M. & Chen, D. *GIM, GRAI integrated methodology – A methodology for designing CIM systems, version 1.0.* Unnumbered report, LAP/GRAI, University of Bourdeaux I, France, 1992.

[55] Duméry, J.J., Faure, J.M., Frachet, J.P., Lampérière, S. & Louni, F. *A tool for the structured modelling of discrete events systems behaviour: the hypergrafcet.* In: Proc. of IMACS-IEEE CESA'96 Multiconference, Symposium on Discrete Events and Manufacturing Systems, pp: 519-524, Lille, France, 1996.

[56] Dye, R. *Labview : a visual data-flow programming language and environment.* Master's thesis, Dept. of Elec. and Comp. Eng. University of Texas at Austin, 1989.

[57] Elmstron, R., Lintulampi, R. & Pezzé, M. *Giving semantics to SA/RT by means of high-level timed Petri Nets.* In: Real-Time Systems, 5, pp: 249-271.

[58] Erickson, K.T. & Hedrick, J.L. *Plantwide process control.* Ed. John Wiley & Sons, 1999. ISBN: 0-471-17835-7.

[59] Faure, J.M., Çouffin, F., Lampérière-Couffin, S. *SAGITAL, un environment d'aide à la conception de grafcets basé sur des méta-modèles.* In: Proc. of Modeling of Reactive Systems, pp: 183-192, 1999.

[60] Ferreiro, R. *Nociones sobre aplicación de PLC's al control de procesos industriales.* Universidade da Coruña, Servicio de publicacións, 1995. ISBN: 84-88301-12-X.

[61] Ferreiro, R., Pardo, X. C., Vidal, J., Coego, J. *Adaptive SFC based supervision algorithm on flexible production systems.* En: Proc. of IFAC Workshop on Control of Industrial Systems, pp: 445-450, Belfort, Francia, 1997. ISBN: 0-08-042907-6.

[62] Ferreiro, R., Vidal, J. Pardo, X. C. *SFC Based FDI with Parity Equations on Hybrid Systems.* En: Proc. of 2nd IMACS Int. Multiconf. on Computational Engineering in Systems Applications. Symposium on Industrial and Manufacturing Systems, pp: 613-617, Nabeul-Hammamet, Tunez, 1998. ISBN: 2-9512309-0-7.

[63] Ferreiro, R., Pardo, X. C., Vidal, J. *Hybrid FDI on Chemical Plants.* En: Proc. of Workshop on On-Line Fault Detection and Supervision in the Chemical Process Industries, pp: 377-381, Solaize-Lyon, Francia, 1998. ISBN: 0-08-043233-6.

[64] Frachet, J.P. & Colombari, G. *Elements for a semantics of the time in Grafcet and dynamic systems using non-standard analysis.* In: Automatique, Productique et Informatique Industrielle, 27(1), pp: 107-125, 1993.

[65] Frachet, J.P. & Lamperiere, S. & Faure, J.M. *The hyperfinite signal: application to the modelling of discrete event systems behaviour.* In: Proc. of IMACS-IEEE CESA'96 Multiconference, Symposium on Discrete Events and Manufacturing Systems, pp. 584-589, Lille, 1996.

[66] Frensel, G. & Bruijin, P.M. *From Grafcet to Hybrid Automata.* In: Proc. of 9th IFAC Symposium on Information Control in Manufacturing, pp: 47-52, Nancy-Metz, France, 1998. Ed. Elsevier Science Ltd. ISBN: 0-08-042928-9.

[67] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. *Design patterns: elements of reusable object-oriented software*, Addison-Wesley, 1995. ISBN: 0-20-163361-2.

[68] Gensym, *G2 Reference Manual, version 4.0.* Gensym Coorporation, 125 Cambridge Park Drive, Cambridge, MA 02140, USA.

[69] Grabow, B.S., Boyle, J.M., Dongarra J.J. & Moler, C. B. editors. *Matrix Eigensystem Routines — EISPACK Guide Extension.* In: Lecture Notes in Computer Science, 51, Springer Verlag, New York, 1977.

[70] Gressier-Soudan, E., Epivent, M., Laurent, A., Boissier, R., Razafindramary, D. & Raddadi, M. *Component oriented control architecture: the COCA project.* In: Microprocessors and Microsystems, 23(2), pp: 95-102, 1999. ISSN: 0141-9331.

[71] Grübel, G., Joos, H., Otter, M. & Finsterwalder, R. *The ANDECS design environment for control engineering.* In: Prepr. of 12th IFAC World Congress, Sydney, Australia, 1993.

[72] Grübel, G., Varga, A., Boom, A. & Geurts, A.J. *Towards a coordinated development of numerical CACSD software: the RASP/SLICOT compatibility concept.* In: Proc. of IEEE Int. Symposium on Computer Aided Control System Design, pp. 499-504, Tucson, 1994.

[73] Guéguen, H. & Bouteille, N. *Extensions of Grafcet to structure behavioural specifications.* In: Control Engineering Practice, 9, pp: 743-756, 2001.

[74] Guillaume, M., Grave, J.M. & Chlique, P. *Formalization of edges for the Grafcet state machine.* In: Proc. of IMACS-IEEE CESA'96 Multiconference, Symposium on Discrete Events and Manufacturing Systems, pp. 579-583, Lille, France, 1996.

[75] Guillemaud, L., Grave, J.M. & Guéguen, H. *Requirements for extending Grafcet to hybrid specifications*, In: Proc. 3rd Int. Conf. Automation of Mixed Processes, pp. 209-215, Reims, France, 1998.

[76] Guillemaud, L. & Guéguen, H. *Extending Grafcet for the specification of control of hybrid sytems*, In: Proc. Int. Conf. Systems, Man and Cybernetics, 1, pp. 171-175, Tokyo, Japan, 1999.

[77] Halepota, A.S., Grant, P.W., & Jobling, C.P. *Design is a document.* In: Proc. of IEE Conference Control, pp: 1-5, Swansea, UK, 1998.

[78] Harel, D. *StateCharts: a visual formalism for complex systems.* In: Science of Computer Programming, 8, 1987.

[79] Hassapis, G., Kotini, I. & Doulgeri, Z. *Validation of a SFC software specification by using hybrid automata.* In: Proc. of 9th IFAC Symposium on Information Control in Manufacturing, pp:65-70, Nancy-Metz, France, 1998. Elsevier Science Ltd. ISBN: 0-08-042928-9.

[80] Hooman, J., Ramesh, S., De Roever, W. *A compositional axiomatization of statecharts.* In: Theoretical Computer Science, 101, pp: 289-335, 1992.

[81] ICAM. *US Air Force Integrated Computer Aided Manufacturing Architecture, Part II, Volume IV-Functional Modeling Manual (IDEF0).* Air Force Materials Laboratory, Wright-Patterson AFB, Ohio 45433, AFWAL-tr-81-4023, 1981.

[82] Ingalls, D. H. *The Smalltalk-76 programming system: Design and implementation.* In: Proc. 5th ACM Symposium on Principles of Programming Languages, pp: 9-16, 1978.

[83] International Electrotechnical Commission. *Graphical symbols for diagrams - Part 12: Binary logic elements.* IEC Standard 60617-12, 1981.

[84] International Electrotechnic Commision. *Preparation of Function Charts for Control Systems. Function Chart Grafcet.* Publicación 60848. Decembro, 1988. Genève, Suisse.

[85] International Electrotechnical Commission. *Grafcet Specification Language for Sequential Function Charts.* IEC Standard 60848, Ed. 2, 2001. Genève, Suisse.

[86] International Electrotechnical Commission. *Programmable Controllers. Part 3: Programming Languages.* Publicación 61131-03. 1993. Genève, Suisse.

[87] Instrument Society of America, ISA-S88.01, *Batch Control, Part 1: Models and Terminology,* Research Triangle Park, NC, 1995.

[88] International Organization for Standardization, ISO/IEC 14882. *Information Technology — Programming Languages — C++,* 1998.

[89] International Organization for Standardization, ISO 9506-1. *Industrial automation systems — Manufacturing Message Specification — Part 1: Service definition,* 1990.

[90] James, J.R. *A survey of knowledge-based systems for computer-aided control system design.* In: Proc. of American Control Conference, Minneapolis, 1987.

[91] Jensen, K. & Rozenberg, G. *High-level Petri Nets.* Springer Verlag, 1991.

[92] Jobling, C. P. *User interface issues and the role of Artificial Intelligence in Computer-Aided Control Engineering.* In: Lecture Notes of ERASMUS intensive course on Application of Artificial Intelligence in Process Control, pp: 303-338. Ed: Boullart, L. Krijgsman, A. and Vingerhoeds, R. A. Pergamon Press, Exeter, England, 1992. ISBN: 0-08-042017-6.

[93] Jobling, C.P. & Varsamidis, T. *The state of the art in CACSD a decade after ECSTASY.* In: Proc. of 8th IFAC symposium on Computer Aided Control Systems Design, Saldford, UK, 2000.

[94] Johnson, C. & Arzén, K. *Graphchart and its relations to Grafcet and Petri Nets.* In: Preprints of 9th symposium on Information Control in Manufacturing, vol. II, pp: 53-58, Nancy-Metz, 1998.

[95] Johnson, C. & Arzén, K. *Batch recipe structuring using high-level Grafchart.* In: Preprints of 13th IFAC World Congress, San Francisco, 1996.

[96] Jones, A.H. *A General Methodology for Converting Petri Nets into Ladder Logic: The TPLL Methodology.* In: Journal of Intelligent Manufacturing, 5, pp. 103-120, 1996.

[97] Kheir, N.A., Aström, K.J., Auslander, D., Cheok, K.C., Franklin, G.F., Masten, M. & Rabins, M. *Control Systems Engineering Education,* In: Automatica, 32, pp: 147-166, 1996.

[98] Klein, S., Frey, G., & Litz, L. *A Petri net based approach to the development of correct logic controllers.* In: Proceedings of the 2nd International Workshop on Integration of Specification Techniques for Applications in Engineering, Grenoble, France, 2002.

[99] Kohn, W., James, J., Nerode, A., Harbison, K. & Agrawala, A. *A hybrid systems approach to computer-aided control engineering.* In: IEEE Control Systems, 15(2), pp: 14-25, 1995.

[100] Kouthon, T., Decotignie, J.D. & Koppenhoefer, S. *On distribution of Grafcet software.* In: Proc. of IMACS-IEEE CESA'96 Multiconference, Symposium on Discrete Events and Manufacturing Systems, pp. 498-506, Lille, France, 1996.

[101] Kristensen, C.H., Andersen, J.H. & Skou, A. *Specification and automated verification of real-time behaviour —a case study.* In: Proc. of 3rd IFAC/IFIP Workshop on Arrchitectures and Algorithms for Real-Time Control, pp: 613-628, Ostend, Belgium, 1995.

[102] Kuikka, S., Tommila, T. & Venta, O. *Distributed Batch Process Management Framework based on Design Patterns and Software Components.* In: Proc. of 14th triennial IFAC World Congress, vol. Q, pp: 263-268, Beijing, China, 1999.

[103] Lessage, J.J. & Roussel, J.M. *Hierarchical approach to Grafcet using forcing order.* In: Automatique, Productique et Informatique Industrielle, 27(1), pp: 25-38, 1993.

[104] Lewin, D. *Design of Logic Systems.* Van Nostrand Reinhold, 1985.

[105] Lewis, R. S. *Programming industrial control systems using IEC 61131-3.* IEE Control Engineering series, The Institution of Electrical Engineers, Exeter, England, 1995. ISBN: 0-85296-827-2.

[106] Lhoste, P., Faure, J.M., Lessage, J.J. & Zaytoon, J. *Comportement temporel du Grafcet.* In: Automatique, Productique et Informatique Industrielle, 31(4), pp: 695-711, 1997.

[107] Little, J.N. Emami-Naeini, A. & Bangert, S.N. *CTRL-C and Matrix Environments for the Computer-Aided Design of Control Systems.* Jamshidi M. and Herget C. J., editors, Computer-Aided Control Systems Engineering, Elsevier Science Publishers, Amsterdam, 1985.

[108] Lobato, V. & Fernández, D. *Fabricación automática de bridas.* E.T.S. de Ingenieros Industriales e Ingenieros Informáticos de Gijón, Universidad de Oviedo, 1999.

[109] Machado, R.J. & Fernandes, J.M., *A Petri Net Meta-Model to develop Software Components for Embedded Systems.* In: Proc. of 2nd Int. Conf. on Application of Concurrency to System Design, pp: 113, Newcastle upon Tyne, UK, 2001.

[110] Maciejowski, J.M. & MacFarlane, A.G. *CLADP: The Cambridge Linear Analysis and Design Programs.* Control Systems Magazine, 1982.

[111] Maffezoni, C., Ferrarini, L. and Carpanzano, E. *Object oriented models for advanced automation engineering.* In: Preprints of 9th Symp. on Information Control in Manufacturing, vol. I, pp: 21-31, Nancy-Metz, France, 1998.

[112] Maffezzoni, C. and Girelli, R. *Moses: Modular Modeling of Physical Systems in an Object-Oriented Database.* In: Mathematics and Computer Modelling of Dynamical Systems, 4, pp. 121-147, 1998.

[113] Mandel, L. and Cengarle, M.V. *On the expressive power of OCL.* In: Proc. of World Congress on Formal Methods in the Development of Computing Systems, volume 1708 of LNCS, pp: 854-874, Toulouse, France, 1999.

[114] Marcé, L. & Le Parc, P. *Modélisation de la sémantique du Grafcet à l'aide de processes synchrones.* In: Proc. of Grafcet'92 conference, pp: 101-110, Paris, France, 1992.

[115] Marcé, L. & Le Parc, P. *Defining the semantics of languages for programmable controllers with synchronous processes.* In: Control Engineering Practice, 1, pp: 79-84, 1993.

[116] Marcé, L., L'Her, D. & Le Parc, P. *Modelling and verification of temporized Grafcet.* In: Proc. of IMACS/IEEE CESA'96 Multiconference, Symposium on Discrete Events and Manufacturing Systems, pax: 783-788, Lille, France, 1996.

[117] Mertins, K., Edeler, H., Jochem, R. & Hofmann, J. *Object oriented modeling and analysis of business processes.* In: Integrated Maufacturing Systems Engineering (P. Ladet an F. Vernadat, eds), pp: 115-28, Chapman&Hall, London, UK, 1995.

[118] Meyer, B. *Eiffel, the Language.* Prentice Hall, 2 edition, 1992.

[119] Moler, C. B. *MATLAB — An Interactive Matrix Laboratory.* Technical Report 369, Department of Mathematics and Statistics, University of New Mexico, 1980.

[120] Morel, G. & Lhoste, P. *Outline for discrete part manufacturing engineering.* In: Proc. of 7th annual European computer conference, pp: 146-155, Paris, France, 1993.

[121] Moreno, S., Peulot, E. *Le GEMMA: modes de marches et d'arrêts, Grafcet de coordination des tâches, Conception des Systèmes Automatisés de Production sûrs.* Editions Casteilla. 1997.

[122] Munro, N. *ECSTASY — A Control System CAD Environment.* In: Proc. of 11th IFAC World Congress on Automatic Control, pp: 13-17, Tallinn, Estonia, 1990.

[123] Musser, D. & Saini, A. *STL Tutorial and Reference Guide.* Addison-Wesley, 1996. ISBN 0-201-63398-1.

[124] Naeger, G. & Rembold, U. *An integrated approach to software system planning and selection based on CIMOSA models.* In: Control Engineering Practice, 3(1), 97-103. 1994.

[125] Naur, P. *Report on the Algorithmic Language algol 60.* In: Communications of the ACM, 6, pp: 1-17, 1960.

[126] OMG. *Common Object Request Broker Architecture, v3.0.* Object Management Group, Inc.

[127] OMG. *UML Semantics, v1.4.* Object Management Group, Inc, September, 2001.

[128] Panetto, H., Lhoste, P., Petin, J.F. & Bierel, E. *Contribution of the Grafcet model to synchrony in discrete events systems modelling.* In: Proc. of IEEE 20th Int. Conf. on Industrial Electronics Control and Instrumentation, 3, pp. 1527-32, Bologne, Italy, 1994. ISBN 0-7803-13291.

[129] Panetto, H. *Stability property of the Grafcet model with (Max, +) algebra.* In: Proceedings of IMACS/IEEE Multiconference CESA'96, Symposium on Discrete Events and Manufacturing Systems, pp: 771-776, Lille, France, 1996.

[130] Pardo, X.C. & Ferreiro, R. *SFC++: a tool for developing distributed real-time control software.* Microprocessors and Microsystems Journal, 23(2), pp: 75-84, 1999.

[131] Pardo, X.C. & Ferreiro, R. *Initial design of a virtual machine to play grafcet models.* In: Preprints of 2nd IFAC/IFIP/IEEE Int. Conf. on Management and Control of Production and Logistics, Grenoble, France, 2000.

[132] Pardo, X.C. & Ferreiro, R. *An object-oriented static metamodel for sequential function charts*. In: Proc. 8th IFAC Symposium on Computer Aided Control Systems Design, pp: 113-118, Salford, UK, 2000.

[133] Pardo, X.C. & Ferreiro, R. *Control of Hybrid Systems with SFC++*. En: Actas do Semanario Anual de Automática, Electrónica Industrial e Instrumentación, Vigo, España, 2003. ISBN: 84-688-3055-6 (en CD).

[134] Pardo, X.C. & Ferreiro, R. *Modeling of Hybrid Sequential Control Systems with SFC++*. In: Proc. of 12th IASTED Int. Conf. on Applied Simulation and Modelling, pp: 663-668, Marbella, Spain, 2003. ISBN: 0-88986-384-9.

[135] Parr, T.J. *Language translation using PCCTS and C++: a reference guide*. Automata Publishing Company, San Jose, CA 95129, USA. 1993. ISBN: 0-9627488-5-4.

[136] Parsaei, H.R. & Sullivan. W.G. (eds) *Concurrent Engineering: contemporary issues and modern design tools*, Chapman&Hall, London. 1993.

[137] Petri, C. *Kommunication mit Automaten*. PhD Thesis. Institut für Instrumentelle Mathematik, Univ. Bonn, 1962.

[138] Pollard, J. *IEC 61131-3: the toolbet is on, but it's not very full*. In: Control Engineering Magazine, February, 1997.

[139] Quérenet, B. *CIMOSA — A European development for enterprise integration*. Part III: Enterprise Integrating Infraestructure. In: Enterprise Integration Modeling (C. Petrie, ed.), pp: 205-15, The MIT Press, Cambridge, MA, 1992.

[140] Rimvall, M. & Cellier, F.E. *A Structural Approach to CACSD*. In: Computer-Aided Control Systems Engineering. Elsevier Science Publishers, Amsterdam, 1985.

[141] Rodd, M. G. & Suski, G. J. *Computer Control Systems: an introduction*. Lecture Notes of ERASMUS intensive course on Application of Artificial Intelligence in Process Control, pp: 205-222. Ed: Boullart, L. Krijgsman, A. and Vingerhoeds, R. A. Pergamon Press, Exeter, England, 1992. ISBN: 0-08-042017-6.

[142] Rogerson, D. *Calling All Members: Member Functions as Callbacks*. MSDN Library, Visual Studio 6.0, Technical Article, 37. 1992.

[143] Rosenbrock, H.H. *Computer-Aided Control System Design*, Academic Press, London, 1974.

[144] Ross, D.T. *Structured Analysis (SA): a language for communicating ideas*. In: IEEE Trans. on Software Engineering, SE-3, pp: 6-15, 1977.

[145] Ross, D.T. *Applications and extensions of SADT*. In: IEEE computer, 18(4), pp: 25-34, 1985.

[146] Rousell, J.M. *Sidoni: de la théorie à la pratique*.

[147] Rousell, J.M. & Lesage, J.J. *Une algèbre de Boole pour l'approche événementielle des systèmes logiques*. In: Automatique, Productique et Informatique Industrielle, 27(5), pp: 541-560, 1993.

[148] Rousell, J.M. & Lesage, J.J. *Validation and verification of Grafcets using State Machine*. In: Proc. of IMACS/IEEE Multiconference CESA'96, Symposium on Discrete Events and Manufacturing Systems, pp: 765-770, Lille, France, 1996.

[149] Roux, O. & Rusu, V. *Du Grafcet au langage réactif Electre*. In: Automatique, Productique et Informatique Industrielle, 28(2), pp: 131-157, 1994.

[150] Rumbaugh, J., Jacobson, I. & Booch, G. *The unified modeling language reference manual.* Ed. Addison-Wesley, 1999. ISBN: 0-201-30998-X.

[151] Sall, K. *XML Family of Specifications: A Practical Guide.* Ed. Addison-Wesley, 2002. ISBN: 0-201-70359-9.

[152] Schwetman, H. D. *CSIM: A C-based, process oriented simulation language,* Tech. Rep. PP-080-85, Microelectronics and Computer Technology Corporation, 1985.

[153] Selic, B., Gullekson, G. & Ward, P. *Real-time object-oriented modeling.* John Wiley & Sons, Inc, 1994. ISBN: 0-471-59917-4.

[154] Sessions, R. *COM and DCOM: Microsoft's Vision for Distributed Objects.* New York, NY: John Wiley & Sons, 1997. ISBN 0-471-19381-X.

[155] Silva, M. & Teruel, E. *A systems theory perspective of discrete event dynamic systems: the Petri Net paradigm.* In: Proc. of IMACS/IEEE CESA'96 Multiconference, Symposium on Discrete Events and Manufacturing Systems, pp: 1-12, Lille, France, 1996.

[156] SLICE, *A Subroutine Library for Control Engineering,* The Numerical Algorithms Group, Oxford, 1987.

[157] Soriano T, Boissier R, Razafindramary D and Raddadi M, *Object-Oriented Analysis of Hybrid Aspects of a Machine Tool.* In: Proc. of IMACS/IEEE CESA'96 Multiconference, Symposium on Discrete Events and Manufacturing Systems, pp: 390-395, Lille, France, 1996.

[158] Spang, H.A.. *The Federated Computer-Aided Control Design System.* In: Computer-Aided Control Systems Engineering, Elsevier Science Publishers, Amsterdam, 1985.

[159] Spechart, F. H., Green, W. L. *A Guide to Using CSMP — the Continuous System Modelling Program.*, pp. 35-39, Prentice Hall, 1976.

[160] Strauss, J.C. *The SCi Continuous System Simulation Language (CSSL),* Simulation, 9(6), 281-303, 1967.

[161] Stroustrup, B. *The C++ programming language, special edition.* Ed: Addison-Wesley, 2000. ISBN: 0-201-70073-5.

[162] Taylor, J. H., Frederick, D. K. Rimvall, C. M. and Sutherland, H. A. *The GE MEAD Computer-Aided Control Engineering Environment.* In: Proc. IEEE Symposium on CACSD, Tampa, FL, 1989.

[163] Taylor, J.H. & Chan, C. *An Expert-Aided implementation interface for Industrial Process Control Systems.* In: Proc. of the IEEE 1999 Mediterranean Control Conference, Haifa, Israel, 1999.

[164] Union Technique de l'Électricité. *Diagramme fonctionnel GRAFCET pour la description des systèmes logiques de commande.* NFC 02, Paris, France, 1982.

[165] Union Technique de l'Électricité. *Function Charts GRAFCET: Extension of basic principles.* NFC 03-191, Paris, France, 1993.

[166] United States Department of Defense. *Reference Manual for the Ada programming language.* DoD, Washington, D.C., January 1983. ANSI/MIL-STD-1815A.

[167] Van der Wal, E. *Acceptance grows for IEC 61131-3 and PLCOpen.* In: Control Engineering Magazine, February, 1997.

[168] Varsamidis, T., Hope, S. & Jobling, C.P. *An object-oriented information model for Computer-Aided Control Engineering*. In: Control Engineering Practice, 4(7), pp: 929-37, 1996.

[169] Vernadat, F. *Enterprise Modelling and Integration. Principles and Applications*. Ed. Chapman&Hall, London, UK. 1996. ISBN: 0-412-60550-3.

[170] Waldner, J. *CIM: principles of computer-integrated manufacturing*. John Wiley & Sons, New York, NY. 1992.

[171] Walker, R., Gregory Jr., C. & Shah. S. *Matrix-X, A Data Analysis, System Identification, Control Design and Simulation Package*. IEEE Control Systems Magazine, 2(4), pp: 30-37, 1982.

[172] Wallén, A. *Using Grafcet To Structure Control Algorithms*. In: Proc. of 3rd European Control Conference, 1995.

[173] Wilczynski, B.K. & Wallace, B.K. *OOPS in real-time control applications*. In: Object oriented software for manufacturing systems, pp: 194-229, Chapman&Hall, London, England, 1992.

[174] Willians, T.J. *The Purdue Enterprise Reference Arquitecture*. In: Computers in industry, 24(2-3), pp:141-158, 1994.

[175] Wirth, N. *The Programming Language Pascal*. In: Acta Informatica, 1, pp: 35-63, 1971.

[176] Wolfram, S. *Mathematica, A System for Doing Mathematics by Computer*, Addisson-Wesley, 1991.

[177] Wonham, W.M. & Ramadge, P.J. *On the supremal controllable sublanguage of a given language*. In: SIAM Journal of Control Optimization, 25, pp: 637-659, 1987.

[178] Wormer, J. & Kleppe A. *The object constraint language, precise modeling with UML*, Addison-Wesley, 1999. ISBN: 0-201-37940-6.

[179] Zaytoon, J. *Specification and design of logic controllers for automated manufacturing systems*. In: Robot & CIM, 12, pp: 353-366, 1996.

[180] Zaytoon J, Richard E, Moughamir S and Angelloz L, *A formalism for Complex Control Systems: Application to a Machine for Training and Re-education of Lower Limbs*. In: Proc. of IMACS/IEEE CESA'96 Multiconference, Symposium on Discrete Events and Manufacturing Systems, pp: 385-389, Lille, France, 1996.

[181] Zaytoon, J. & Villermain-Lecolier, G. *Two methods for the engineering of manufacturing systems*. In: Control Engineering Practice, 5(2), pp: 185-198, 1997.

[182] Zaytoon, J., Carré-Ménétrier, V., Niclet, M. & De Loor, P. *On the recent advances in Grafcet*. In: Preprints of the IFAC Workshop on Manufacturing Systems: Modelling, Management and Control, pp: 419-424, Vienna, Austria, 1997.

[183] Zaytoon, J., Ndjab, C. & Carré-Ménétrier. *On the synthesis of Grafcet using the supervisory control theory*. In: Proc. of IFAC conference, pp: 391-396, Belfort, France, 1997.

[184] Zaytoon, J., Ndjab, C. & Roussel, J.M. *On the supremal controllable Grafcet of a given Grafcet*. In: Proc. of 2nd IMACS MATHMOD conf., pp: 371-376, Vienna, Austria. 1997.